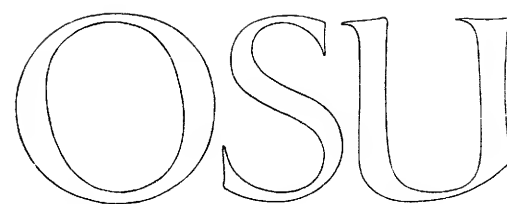


BASIC User's Manual

May 1972



COMPUTER CENTER

Oregon State University
Corvallis, Oregon 97331

BASIC User's Manual

ccm-71-08

Computer Center
Oregon State University
Corvallis, Oregon 97331

May 1972

TABLE OF CONTENTS

	<u>Page</u>
INTRODUCTION	1
CHAPTER 1 - A BASIC PRIMER	3
An Example	3
Formulas	8
Numbers	11
Variables	11
Relational Operators	12
Loops	13
Lists and Tables	16
Errors and Debugging	19
CHAPTER 2 - SUMMARY OF ELEMENTARY BASIC STATEMENTS	25
Expressions	25
Arithmetic Expressions	25
Relational Expressions	27
Logical Expressions	28
Simple Assignment (LET)	28
GO TO	29
ON - GO TO	29
IF - THEN/IF - GO TO	30
FOR and NEXT	30
READ and DATA	32
INPUT	33
PRINT	34
DIM	35
Functions	35
INT	35
RND	36
SGN	38
DEF	38
END	39
CHAPTER 3 - ADDITIONAL BASIC STATEMENTS	41
GOSUB and RETURN	41
STOP	43
REM	43
RESTORE	44
CHAPTER 4 - ADVANCED BASIC	45
Alphanumeric Data and String Manipulation	45
DIM	45
LET	46
READ and DATA	46

INPUT	46
PRINT	47
IF - THEN/IF - GOTO	47
More About Printing	48
PRINT	48
TAB	50
Rules for Printing Numbers	50
PRINT USING and Image Statements	51
Integer Fields	52
Decimal Fields	53
Exponential Fields	53
Alphanumeric Fields	54
Literal Fields	54
General Rules	55
CHAPTER 5 - FILE INPUT/OUTPUT	57
Data Files	57
File Opening	57
File Reading	58
File Writing	58
End-of-File Check	58
File Restoring	59
File Scratching	59
File Closing	59
Characteristics of BASIC Data Files	60
CHAPTER 6 - MATRICES	61
MAT READ and MAT PRINT	62
Matrix Arithmetic	62
Scalar Multiplication	63
Identify Matrix	63
Matrix Transposition	64
Matrix Inversion	64
Matrix ZER and CON Functions	64
Dimensioning	65
Sample Matrix Programs	66
Matrix Program Example 1	67
Matrix Program Example 2	68
CHAPTER 7 - EXAMPLES OF ADVANCED BASIC PROGRAMS	71
Inventory Problem	71
Value of E to 1000 Places	74
APPENDICES	
APPENDIX A - TELETYPE OPERATION	79
APPENDIX B - FORMS OF BASIC STATEMENTS	81

APPENDIX C - ERROR MESSAGES	85
Compilation Errors	85
Command Mode Errors	86
Execution Errors	86
APPENDIX D - COMPARISON ORDER FOR BASIC CHARACTERS	91

INTRODUCTION

A program is a set of directions that is used to tell a computer how to provide an answer to some problem. It usually starts with the given data, contains a set of instructions to be carried out in a certain order, and ends with a set of answers.

Any program must meet two requirements before it can be carried out: first, it must be presented in a form that the computer understands; and second, it must be completely and precisely stated.

The computer's basic language consists of elementary INSTRUCTIONS such as add, subtract, punch a card, or shift a number left or right. The problem-solving procedure must therefore be TRANSLATED into simple instructions that the computer is capable of obeying. This translation, which is called programming or coding, can be carried out entirely by a human being, or the computer may assist in the process by use of a compiler.

A compiler is a large set of computer instructions which can accept a problem-solving procedure, written in a form resembling the language of the procedure, and produce from it the proper elementary machine instructions that will solve the problem. The programming language to use is BASIC, Beginner's All-purpose Symbolic Instruction Code. BASIC is precise, simple, and easy to understand.

An introduction to writing a BASIC program is given in Chapters 1 and 2, which includes all that you need to know to write a variety of useful and interesting programs. Chapters 3-6 deal with more advanced techniques. The appendices contain a variety of reference materials.

CHAPTER 1 - A BASIC PRIMER

AN EXAMPLE

The following example is a complete BASIC program for solving a system of two simultaneous linear equations in two variables:

$$ax + by = c \qquad dx + ey = f$$

and then solving two systems, each differing from this system only in the constants c and f. If $ae - bd$ is not equal to zero, you should be able to solve this system to find that:

$$x = \frac{(ce - bf)}{(ae - bd)} \quad \text{and} \quad y = \frac{(af - cd)}{(ae - bd)}$$

If $ae - bd$ is equal to zero, either there is no solution or there are infinitely many, but there is no unique solution. If you are rusty on solving such systems, take our word for it that this is correct. For now, we simply want you to understand the BASIC program for solving this system.

Study the following program carefully -- the purpose of most lines in the program is self-evident -- and then read the commentary and explanation.

```
100 READ A,B,D,E
110 LET G=A*E-B*D
120 IF G=0 THEN 180
130 READ C,F
140 LET X=(C*E-B*F)/G
150 LET Y=(A*F-C*D)/G
160 PRINT X,Y
170 GO TO 130
180 PRINT "NO UNIQUE SOLUTION"
190 DATA 1,2,4
200 DATA 2,-7,5
210 DATA 1,3,4,-7
999 END
```


Notice that the program consists of several lines or STATEMENTS, headed by a LINE NUMBER.

The line numbers also serve to specify the order in which the statements are to be performed by the computer. This means that your program may be typed in any order. Before the program is run, the computer sorts and edits the program, putting the statements into the order specified by their line numbers. This editing process also facilitates correcting and changing (debugging) programs, as explained later.

Note that each statement starts, after its line number, with an English word. The word denotes the type of the statement. There are several types of statements in BASIC, nine of which are discussed in chapters 2 and 3. Seven of these nine appear in the sample program we are now considering.

Note also that, although it is not obvious from the program, blanks have no significance in BASIC statements, except in messages that are to be printed out, as in line number 180. Blanks may or may not be used to make a program more readable. Statement 100 could have been typed as 100READA,B,D,E and statement 110 as 110LETG=A*E-B*D.

The first statement, 100, is a READ statement. It must be accompanied by one or more DATA statements. When the computer encounters a READ statement while executing the program, it causes the variables listed after the word READ to be given values according to the next available numbers in the DATA statements. In the sample program, A in statement 100 is read and assigned the value 1 from statement 190, similarly B is 2, and D is 4. At this point, the available data in statement 190 has been exhausted, but there is more in statement 200.

Next go to statement 110, a LET statement, where a formula to be evaluated is first encountered. (The asterisk, *, is used to denote multiplication.) In this statement the computer is directed to find the value of $AE - BD$, and to call the result G.

In general, a LET statement shows the principal way calculations on numbers are performed. The formula on the right hand side of the equal sign is evaluated by using the current values of the variables that appear. Addition and subtraction are denoted with + (plus) and - (minus) signs respectively; multiplication, division, and exponentiation by * (asterisk), / (slash), and ↑ (up arrow) respectively.

If G is equal to zero, the system has no unique solution. Therefore, in line 120, the computer is asked, whether G is equal to zero. If the computer finds a YES answer to the question, it is directed to go to line 180. Line 180 tells it to print out NO UNIQUE SOLUTION. From this point, it would go to the next statement. But lines 190, 200, and 210 give it no instructions, since DATA statements are not executed, therefore it goes to line 999, which directs it to END the program.

If the answer to the question "Is G equal to zero?" is NO, as it is in the sample program, the computer simply goes to the next statement, in this case statement 130. (An IF--THEN statement tells the computer where to go if the IF condition exists, but to go on to the next statement if it does not exist.) Statement 130 directs the computer to read the next two entries from the DATA statement -- in this case -7 and 5, both in statement 200 -- and to assign them to C and F respectively. The computer is now ready to solve the system:

$$X + 2y = -7 \qquad 4x + 2y = 5$$

In statements 140 and 150, the computer is directed to find the values of X and Y according to the formulas provided. Note that parentheses must be used to show that CD - BD is divided by G. Without parentheses, only BF would be divided by G, and the computer would find:

$$x = \frac{ce - bf}{g}$$

The computer is told in line 160 to print the two values computed, those of X and Y. Having done so, it moves on to line 170, where it is directed back to line 130. If there are additional numbers in the DATA statements, as there are here in 210, the computer is told in line 130 to take the next number and assign it to C, and the one after that to F. The computer is now ready to solve the system:

$$x + 2y = 1 \qquad 4x + 2y = 3$$

As before, it finds the solution in 140 and 150, prints out the values in 160, and then is directed in statement 170 to go back to 130.

In line 130 the computer reads two more values, 4 and -7, which it finds in line 210. It then solves the system:

$$x + 2y = 4 \qquad 4x + 2y = -7$$

and prints out the solutions. It is directed back again to 130, but there are no more pairs of numbers available for C and F in the DATA statements. The computer therefore informs you that it is out of data, by printing OUT OF DATA 130, and stops.

Let us look at the importance of the various statements. For example, what would have happened if line number 160 had been omitted? The computer would have solved the equations three times and then told us it was out of data. However, since it was not told to show us (PRINT) the answers, it would not do so, and the solutions would be the computer's secret.

What would have happened if line 120 had been omitted? In the problem just solved, nothing. But if G were equal to zero, the computer would have had the impossible task of dividing by zero in 140 and 150, error messages DIVIDE FAULT 140 and DIVIDE FAULT 150 would have been printed. Suppose statement 170 had been omitted. The computer would have solved the first system, printed out the values X and Y, and then gone on to line 180 as

directed; it would print out NO UNIQUE SOLUTION, and then stop.

A natural question that may arise is, why the selection of line numbers. The particular choice of line numbers is arbitrary. The only requirement is that statements be numbered in the order that the computer is to follow in executing the program. The statements could have been numbered 1, 2, 3, 4, ..., 13; however, this is not recommended. The statements would normally be numbered 100, 110, 120, ..., 999. The numbers are placed such a distance apart so that later additional statements may be inserted easily. For example, if it is found that two statements belonging between those numbered 140 and 150 were left out, they can be given any two numbers between 140 and 150, say 144 and 146. In the editing and sorting process, the computer puts them in the correct place.

Another question that may arise has to do with the placing of the elements of data in the DATA statements: why place them as they are in the sample program. The choice is arbitrary. The numbers need only be arranged in the order that they are to be read -- the first for A, the second for B, the third for D, the fourth for E, the fifth for C, and so on. In place of the three statements numbered 190, 200, 210, the data may have been entered as:

```
195 DATA 1,2,4,2,-7,5,1,3,4,-7
```

or, perhaps more logically:

```
190 DATA 1,2,4,2
200 DATA -7,5
210 DATA 1,3
220 DATA 4,-7
```

by putting the coefficients in the first DATA statement and the three pairs of righthand constants in the following DATA statements.

Finally, every BASIC program must have an END statement. It must be the last (highest numbered) statement in the program,

in this case it is statement 999. This statement marks the end of the program, and it is also used to stop computation.

After inserting the program, type RUN followed by a carriage return. Up to this point the computer stores the program and does nothing else with it. It is the command RUN that directs the computer to execute the program.

The sample program and the resulting printout are shown now as they appear on the printer.

```
110 LET G=A*E-B*D
120 IF G=0 THEN 180
130 READ C,F
140 LET X=(C*E-B*F)/G
150 LET Y=(A*F-C*D)/G
160 PRINT X,Y
170 GO TO 130
180 PRINT "NO UNIQUE SOLUTION"
190 DATA 1,2,4
200 DATA 2,-7,5
210 DATA 1,3,4,-7
999 END
RUN
```

4	-5.50000
0.666667	0.166667
-3.66667	3.83333

OUT OF DATA 130

FORMULAS

The computer can carry out a great many operations. It can add, subtract, multiply, divide, extract square roots, raise a number to a power, find the sine of a number or an angle measured in radians, and so on.

The computer calculates by evaluating formulas that are supplied in a program. The formulas are similar to those used in ordinary mathematical calculation, except that each BASIC formula must be written on a single line. Five arithmetic operators can be used

to write a formula. They are listed in the following table.

SYMBOL	EXAMPLE	MEANING
+	$A + B$	Addition. Add B to A
-	$A - B$	Subtraction. Subtract B from A
*	$A * B$	Multiplication. Multiply B by A
/	A / B	Division. Divide A by B
↑	$A \uparrow 2$	Raise to the power. Find $X \uparrow 2$

Parentheses may be used to group together a set of operations that must be calculated as a unit. First, however, it is necessary to be familiar with the hierarchy of operations performed by the computer.

For example, if $A + B * C \uparrow D$ is typed, the computer first raises C to the power D, then multiplies this result by B and then adds A to the resulting product. This is the usual convention for $A + BC \uparrow D$. If this is not the intended order, parentheses must be used to indicate a different order.

Suppose it is the product of B and C that is to be raised to the power D. The statement is written $A + (B * C) \uparrow D$. Or, if we want to multiply A + B by C to the power D, then the statement is $(A + B) * C \uparrow D$. To add A to B, multiply their sum by C, and raise the product to the power D, write $((A + B) * C) \uparrow D$.

The order of priorities for calculating is according to the following rules.

1. The formula inside the parentheses is computed before the enclosed quantity is used in further calculations.
2. In the absence of parentheses in a formula that includes addition, multiplication, and the raising of a number to a power, the computer first raises the number to the power, then does the multiplication, and finally the addition. Division has the same priority as multiplication, and subtraction the same as addition.

3. In the absence of parentheses in a formula that includes only multiplication and division (or only addition and subtraction), the computer calculates from left to right.

The rules are illustrated in the sample program already considered. The rules also tell us that the computer, given $A-B-C$, subtracts B from A and then C from their difference. Given $A/B/C$, it divides A by B and then that quotient by C . Given $A \uparrow B \uparrow C$, the computer raises the number A to the power B and then raises the resulting number to the power C . If there is ever any question about the priority, insert more parentheses to avoid possible ambiguities.

In addition to the five arithmetic operations, the computer can evaluate several mathematical functions. The functions are given special letter names, as shown in the following table.

FUNCTION	MEANING	
SIN(X)	Find the sine of X	X interpreted as a number, or as an angle measured in radians
COS(X)	Find the cosine of X	
TAN(X)	Find the tangent of X	
ATN(X)	Find the arctangent of X	
EXP(X)	Find $E \uparrow X$	
LOG(X)	Find the natural logarithm of X ($\ln X$)	
ABS(X)	Find the absolute value of X ($ X $)	
SQR(X)	Find the square root of X (\sqrt{X})	

Three other mathematical functions are available in BASIC: INT, RND, and SGN. These are reserved for explanation in Chapter 2.

In place of X , any formula or number in parentheses may be substituted following any of the functions listed above. For example, to find the square root of $4 + X \uparrow 3$, write $SQR(4 + X \uparrow 3)$, or the arctangent of $3X - 2E \uparrow X + 8$, write $ATN(3 * X - 2 * EXPT(X) + 8)$.

NUMBERS

Since numbers and variables have already been mentioned it is important to understand how to write numbers for the computer and what variables are allowed.

A number may be positive or negative, and may contain as many as eleven digits, but it must be expressed in decimal form. For example, all of the following are numbers in BASIC:

2 -3.675 123456789 -.987654321 483.4156

The following are not numbers in BASIC:

14/3 $\sqrt{7}$.000123456789

The first two are formulas, but not numbers. The last example has more than eleven digits. The computer may be asked to calculate the decimal expansion of 14/3 or $\sqrt{7}$, and to do something with the resulting number, but neither can be included in a list of DATA.

Additional flexibility is provided by using the letter E, which stands for "times ten to the power." Using E, .00123456789 may be written in several forms acceptable to the computer: .123456789E-2 or 123456789E-11 or 1234.56789E-6. Ten million may be written as 1E7 and 1969 as 1.969E3. A number may not be written as E7, instead it is written as 1E7 to indicate that it is 1 that is multiplied by 10. The power of ten may range between +308 and -308.

VARIABLES

A variable in BASIC is denoted by any letter, or by any letter followed by a single digit. The computer interprets E7 as a variable, along with A, X, N5, 10, and O1. A variable in BASIC represents a number, usually one that is unknown to the

programmer at the time the program is written. Variables are given or assigned values by LET, READ, and INPUT statements. All variables have the initial value of zero, so that if the starting value of a variable is to be zero, it is not necessary to assign it that value. (Another kind of variable, the string variable, is discussed later in Chapter 4.)

Although the computer does little in the way of correcting during computation, it will issue error messages to indicate that an absolute value was not used. For example, if the square root of -7 or the logarithm of -5 is requested, the computer gives the square root of 7 or the logarithm of 5 noting that the square root or the logarithm of a negative number was requested.

RELATIONAL OPERATORS

Three other mathematical symbols, symbols of relation, are available in BASIC to indicate any of six standard relations. These are used in IF--THEN statements, where values must be compared. The six possible relations are shown in the following table. See Chapter 2 for more information on relational operators and their use in IF--THEN statements.

SYMBOL	EXAMPLE	MEANING
=	A = B	is equal to. A is equal to B.
<	A < B	is less than. A is less than B.
<=	A <= B	is less than or equal to. A is less than or equal to B.
>	A > B	is greater than. A is greater than B.
>=	A >= B	is greater than or equal to. A is greater than or equal to B.
<>	A <> B	is not equal to. A is not equal to B.

LOOPS

Often a program may be needed where one or more parts are traversed not just once, but a number of times, perhaps with slight changes each time. In order to write the simplest program, one in which the part to be repeated is written just once, the programming device is known as a LOOP.

The use of loops can be illustrated by two programs for the simple task of printing a table of the first 100 positive integers together with the square root of each. Without a loop, the program would be 101 lines long:

```
100 PRINT 1,SQR(1)
105 PRINT 2,SQR(2)
110 PRINT 3,SQR(3)
      .
      .
      .
590 PRINT 99,SQR(99)
595 PRINT 100,SQR(100)
600 END
```

With the following program, using one type of loop, the same table may be generated with only 5 lines of instructions instead of 101.

```
100 LET X =1
110 PRINT X,SQR(X)
120 LET X=X+1
130 IF X<=100 THEN 110
999 END
```

Statement 100 gives the value of 1 to X, which initializes the loop. Line 120 increases the value of X by 1, to 2. Line 130 asks whether X is less than or equal to 100--a YES answer directs the computer back to line 110. Here it prints 2 and $\sqrt{2}$, and goes to 120. Again X is increased by 1, now to 3, and at 130 it goes back to 110. This process is repeated -- line 110 (prints 3 and

$\sqrt{3}$), line 120 ($X = 4$), line 130 (since 4 is less than or equal to 100 go back to line 110), and so on -- until the loop has been traversed 100 times. Then X becomes 101. The computer now finds a NO answer to the question in line 130 (X is greater than 100, not less than or equal to 100). It therefore does not return to 110 but moves on to line 999, and ends the program.

Note: In line 120, the variable X appears on both sides of the equal (=) sign. The = sign stands for replacement (but not equivalent to). This is an important distinction, since it is not a statement of algebraic equality. The proper interpretation is "evaluate the formula on the right, and let this value be assigned to the variable on the left."

All loops contain four elements: initialization (line 100 in the program), the body (line 110), modification (line 120), and an exit test (line 130). Because loops are so important, BASIC provides a pair of statements that specify a loop even more simply than the previous program. They are the FOR and NEXT statements. Their use is illustrated in this program.

```
100 FOR X=1 TO 100
110 PRINT X,SQR(X)
120 NEXT X
999 END
```

which performs exactly the same operation as the two previous programs. In line 100, X is set equal to 1, and a test is set up, exactly as in line 130 above. Line 120 causes X to be increased by 1, and also carries out the test to decide whether to return to line 110 or to continue. Thus lines 100 and 120 take the place of lines 100, 120 and 130 in the previous program -- and they are easier to use.

Note that the value of X is increased by 1 each time through the loop. If a different increase is needed, it may be specified by writing:

```
100 FOR X=1 TO 100 STEP 5
```

and the computer assigns 1 to X on the first time through the loop, 6 to X on the second time through, 11 on the third time, and 96 on the twentieth time. Another step of 5 would take X beyond 100, therefore the program prints 96 and its square root and terminates. The step value may be either positive or negative. The first table printed may be obtained in reverse order by writing line 100 as:

```
100 FOR X=100 TO 1 STEP -1
```

In the absence of a STEP instruction, a step size of +1 is assumed.

More complicated FOR statements are allowed. The initial and final value, and the step size may all be formulas of any complexity. For example, if N and Z have been specified earlier in the program, the following may be written:

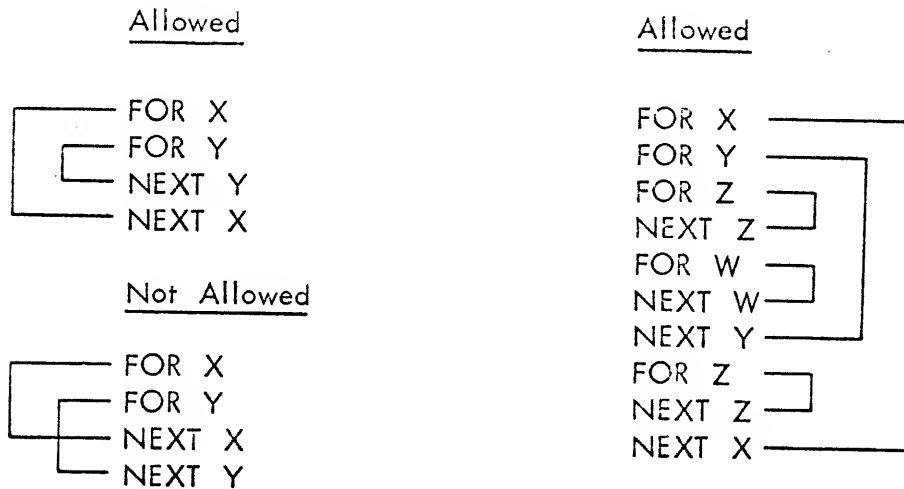
```
250 FOR X=N+7*Z TO (Z-N)/3 STEP (N-4*Z)/10
```

For a positive step size, the loop continues as long as the control variable is less than or equal to the final value. For a negative step size, the loop continues as long as the control variable is greater than or equal to the final value.

If the initial value is greater than the final value (or less than for a negative step size), the body of the loop will not be executed even once. The computer immediately passes to the statement following the NEXT. For example, the following program adds up the first N integers to give the correct result 0, when N is 0.

```
100 READ N
110 FOR K=1 TO N
120 LET S=S+K
130 NEXT K
140 PRINT S
150 GO TO 100
160 DATA 2,10,0
999 END
```

It is often useful to have loops within loops. These are called NESTED LOOPS. They can be expressed with FOR and NEXT statements, but must actually be nested and must not cross, as the following examples illustrate:



LIST AND TABLES

In addition to the ordinary numeric variables used in BASIC, there are variables that may be used to designate the elements of a list or table. Ordinarily a subscript or a double subscript is used at this time as for the coefficients of a polynomial (a_1, a_2, \dots) or the elements of a matrix $b_{i,j}$. The variables used in BASIC consist of a single letter, which is called the name of the list or table, followed by the subscripts in parentheses. For the coefficients of the polynomial, write $A(1), A(2)$, and so on; for the elements of the matrix, write $B(1,1), B(1,2)$, and so on.

The list $A(1), \dots, A(10)$ may be entered into a program very simply with four lines:

```

100 FOR I=1 TO 10
110 READ A(I)
120 NEXT I
130 DATA 2,3,-5,7,2.2,4,-9,123,4,-4

```

No special instructions to the computer are necessary if a subscript no greater than 10 occurs. For large subscripts, a dimension (DIM) statement must be used, to tell the computer to save extra space for the list or table. When in doubt, indicate a larger dimension than you expect to use. For example, if a list of 15 numbers entered is desired, write:

```

100 DIM A(25)
110 READ N
120 FOR I=1 TO N
130 READ A(I)
140 NEXT I
150 DATA 15
160 DATA 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47

```

Statements 110 and 150 may have been eliminated by writing 120 as FOR I=1 TO 15. Using this form allows the list to be lengthened by changing only statement 150, as long as the number of elements in the list does not exceed 25.

To enter a 3 x 5 table into a program write:

```

100 FOR I=1 TO 3
110 FOR J=1 TO 5
120 READ B(I,J)
130 NEXT J
140 NEXT I
150 DATA 2,3,-5,-9,2
160 DATA 4,-7,3,4,-2
170 DATA 3,-3,5,7,8

```

A table may be entered with no dimension statement, and the computer will handle all the entries from B(1,1) to B(10,10). But if you try to enter a table with a subscript greater than 10, without a DIM statement, an error message is received saying: SUBSCRIPT ERROR. This can be easily corrected by entering, for example, the line:

```

95 DIM B(20,30)

```

which reserves space for a 20 by 30 table.

The single letter denoting a list or table name may also be used to denote both a list and a table in the same program.

The form of the subscript is quite flexible. You might have the list $B(I + K)$ or the table items $B(I,K)$ or $Q(A(3,7),B - C)$.

The following sample program uses both a list and a table. The program computes the total sales of each of five salesmen, each of whom sells the same three products.

```
100 FOR I=1 TO 3
110 READ P(I)
120 NEXT I
130 FOR I=1 TO 3
140 FOR J=1 TO 5
150 READ S(I,J)
160 NEXT J
170 NEXT I
180 FOR J=1 TO 5
190 LET S=0
200 FOR I=1 TO 3
210 LET S=S+P(I)*S(I,J)
220 NEXT I
230 PRINT "TOTAL SALES FOR SALESMAN ";J,"$";S
240 NEXT J
250 DATA 1.25,4.30,2.50
260 DATA 40,20,37,29,42
270 DATA 10,16,3,21,8
280 DATA 35,47,29,16,33
999 END
```

RUN

```
TOTAL SALES FOR SALESMAN 1 $ 180.500
TOTAL SALES FOR SALESMAN 2 $ 211.300
TOTAL SALES FOR SALESMAN 3 $ 131.650
TOTAL SALES FOR SALESMAN 4 $ 166.550
TOTAL SALES FOR SALESMAN 5 $ 169.400
```

The list P gives the price per item of each of the three products. Table S tells how many items of each product each man sold. Product number 1 sells for \$1.25 per item, number 2 for \$4.30 per item, and number 3 for \$2.50 per item. Salesman number 1 sold 40 items of the first product, 10 of the second, and 35 of the third, and so on. The program reads in the price list in lines 100, 110, and 120, using data in line 250. It reads in the sales table in lines 130 to 170, using data in lines 260 and 280. The

same program could be used again, modifying only line 250 if the prices change, and only lines 260 to 280 to enter the sales in another month.

The sample program did not need a dimension statement, since the computer automatically saves enough space to allow subscripts to run from 1 to 10. A DIM statement is normally used when it is necessary to save more space. But in a long program, requiring many small tables, DIM may be used to save less space for tables, in order to leave more room for the program.

Since a DIM statement is not executed, it may be entered into the program on any line before END. It is convenient, though, to place DIM statements near the beginning of the program.

ERRORS AND DEBUGGING

Occasionally the first run of a new program will be free of errors and give the correct answers. Usually, though, errors must be corrected before the program runs correctly. Errors are of two types: errors of form that prevent the running of the program, and logical errors in the program that cause the computer to produce either incorrect answers or no answer at all.

Errors of form cause error messages to be printed. The various error messages are listed and explained in Appendix A. Logical errors are often much harder to find, particularly when the program gives answers that are nearly correct. In either case, after the errors are found, they can be corrected by changing, inserting, or deleting lines from the program. A line is changed by typing it correctly with the same line number. A line is inserted by typing it with a line number between those of two existing lines. A line is deleted by typing its line number and pressing the RETURN key. Notice that a line can be inserted only if the original line numbers are not consecutive. For this reason, most programmers begin by using line numbers that are multiples of five or ten, but that is a matter of choice.

Corrections may be made at any time when they are noticed, either before or after a run. Since the computer sorts lines and arranges them in order, a line may be retyped out of sequence. Simply retype the corrected line with its original line number.

As with most problems in computing, the process of finding errors (or bugs) in a program, and correcting (or debugging) it, is best illustrated by an example. Consider the problem of finding the value of X between 0 and 3 for which the sine of X is a maximum, and printing out this value of X and the value of its sine. If you have studied trigonometry, you know that $\pi/2$ is the correct value of X , but we shall use the computer to test successive values of X from 0 to 3. Intervals of .1, then of .01, and finally of .001 will be used.

Thus, the computer is directed to calculate the sine of 0, .1, .2, .3, ..., 2.8, 2.9, and of 3, and to determine which of these 31 values is the largest. It will do so by testing $\text{SIN}(0)$ and $\text{SIN}(.1)$ to see which is larger, and calling the larger of these two numbers M . Then it will pick the larger of M and $\text{SIN}(.2)$, and call it M . This number will be checked against $\text{SIN}(.3)$, and so on down the line. Each time a larger value of M is found, the value of X is stored in $X0$. When the computer finishes the series, M will have the value of the largest of the 31 sines, and $X0$ will be the argument that produced that largest value. It will then repeat the search, this time checking the 301 numbers 0, .01, .02, .03, ..., 2.98, 2.99, and 3, finding the sine of each and checking to see which sine is the largest. Finally, it will check the 3001 numbers 0, .001, .002, .003, ..., 2.999, and 3, to find which has the largest sine. At the end of each of the three searches, the computer is to print three numbers: the value $X0$ that has the largest sine, the sine of that number, and the interval of search.

Assume the program is the following:

```

100 READ D
110 LET XO=0
120 FOR X=0 TO 3 STEP D
130 IF SIN(X) = M THEN 200
140 LET XO=X
150 LET M=SIN(XO)
160 PRINT XO,X,D
170 NEXT XO
180 GO TO 110
190 DATA .1,.01,.001
999 END

```

The entire sequence is illustrated with explanatory comments on the righthand side.

```

?110 LWR XO=0
?120 FOR X=0 TO 3 STEP D
?130 IF SIN (X)<=M THEN 200
?140 LET XO=X
?150 LET M=SIN(X)
?160 PRINT XO,X,D
?170 NEXT XO
?180 GOTO 110
?110 LET XO=0
?190 DATA .1,.01,.001
?999 END
?RUN
ILLEGAL FORMULA 160
NOT MATCH WITH FOR 170
LABEL IS UNDEFINED 130

```

After typing line 180 it was noticed that LET was mistyped in line 110. Thus it is retyped this time correctly.

When the first error message is received, line 160 is inspected and it is found that XO was used instead of X for a variable. The next error message refers to line 170 where the variables were mixed. This is corrected by changing line 170. The last error message points out that line 130 refers to a line that does not exist. Line 130 is retyped with the correct statement number.

```

?160 PRINT XO,X,D
?170 NEXT X
?130 IF SIN(X)<=M THEN 170
?RUN
0.10000      0.10000      0.10000
0.20000      0.20000      0.10000
0.30000      0.30000      0.10000
0.40000      0.

```

This is obviously incorrect. Every value of X is being printed. Stop the printing by pressing break. Notice that SIN (0) is compared with M on the first time through the loop, but M has not yet been assigned a value. By changing line 110, M is given the value of -1; previously X0 had been initialized instead of M.

```
?110 LET M=-1
```

```
?RUN
```

0	0	0.10000
0.10000	0.10000	0.10000
0.20000	0.20000	0.10000
0.30000		

The same table is being printed as before. It is printing out values each pass through the loop. This is remedied by deleting line 160 and retyping it as line 175, which puts the PRINT statement outside the loop. Also, it is noticed that M and not X is to be printed.

```
?160
```

```
?175 PRINT X0,M,D
```

```
?RUN
```

1.60000	0.99957	0.10000
1.60000	0.99957	0.

Notice that the same results are again being received. The case for D=.1 is being repeated. Another look at the program shows that 180 sent us back to line 110 to repeat the operation rather than back to line 100 to pick up a new value for D. We also decide to put in headings for our columns using a PRINT statement.

```
?180 GOTO 100
?95 PRINT "X VALUE","SINE",RESOLUTION"
?RUN
ILLEGAL FORMULA 95
?95 PRINT "X VALUE","SINE","RESOLUTION"
?RUN
```

There is a missing quotation mark. Correct and try again.

X VALUE	SINE	RESOLUTION
1.60000	0.99957	0.10000
1.57000	1.00000	1.00000E-02
1.57100	1.00000	1.00000E-03

OUT OF DATA 100

Exactly the desired results. Of the 31 numbers 0, .1, .2, .3, ..., 2.8, 2.9, 3, it is 1.6 that has the largest sine, namely 0.99957. Similarly for the finer subdivisions.

Having changed so many parts of the program, a list of the corrected program is asked for.

```
?LIST
95 PRINT "X VALUE","SINE","RESOLUTION"
100 READ D
110 LET M=-1
120 FOR X=0 TO 3 STEP D
130 IF SIN(X)<=M THEN 170
140 LET X0=X
150 LET M=SIN(X)
170 NEXT X
175 PRINT X0,M,D
180 GOTO 100
190 DATA .1,.01,.001
999 END
?FILE,MAXSIN
```

The program is saved for later use. This should not be done unless you expect to use the program later. The name of the file is MAXSIN.

In solving this problem, two common programming utilities have been used: PRINT and LIST. A PRINT statement may be inserted in the program to confirm what the computer is actually calculating. For example, the statement 155 PRINT M may have been inserted to direct the computer to print the value of M. After several corrections have been made to a program, it is often desirable to see a copy of the entire program. Simply type LIST, and the computer prints out the program in its current form.

CHAPTER 2 - SUMMARY OF ELEMENTARY BASIC STATEMENTS

This section gives a concise description of each of the types of BASIC statements most useful in writing the simpler kinds of BASIC programs. For each form, assume a line number and use underlining to denote a general type. Thus, variable means any variable, which is a single letter, possibly followed by a single digit.

EXPRESSIONS

An expression is a constant, a simple or subscripted variable, a function, or any combination of these separated by operators and parentheses. Expressions can be arithmetic which have numerical values, or logical and relational which have truth values. Each type of expression has an associated group of operators and operands.

ARITHMETIC EXPRESSIONS

The following operators are used in arithmetic expressions:

<u>Symbol</u>	<u>Function</u>
+	Addition
-	Subtraction
*	Multiplication
/	Division
↑	Exponentiation

Arithmetic operands are:

Constants

Variables (simple, subscripted, or strings)

Functions

Examples:

```
A
3.14159265
B+16.4832
(A-B(I,J))
G*C(J)+4.1/(Z(J,3*K))*SIN(V)
```

Rules

1. In an arithmetic expression do not use adjacent arithmetic operators, $X \text{ OP OP } Y$, or adjacent arithmetic elements, $A(B+C)D$.
2. If X is an expression, then (X) , $((X))$, etc., are expressions.
3. If X and Y are expressions, the following are expressions:

$X+Y$	X/Y
$X-Y$	$X*Y$

4. There is no implied multiplication. $X(Y)$ does not imply $X*(Y)$. $2(X)$ does not imply $2*(X)$.
5. The left and right square brackets, $[$ and $]$ may be used interchangeably with the left and right parentheses, $($ and $)$ any place in BASIC.

Order of Evaluation

The hierarchy of arithmetic operations is:

\uparrow	Exponentiation	Class 1
$/$	Division	Class 2
$*$	Multiplication	Class 2
$+$	Addition	Class 3
$-$	Subtraction	Class 3

In an expression with no parentheses or within a pair of parentheses, evaluation proceeds from left to right, if unlike classes of operators appear. The first operator of such an expression is compared against the second. If the first operator takes precedence over the second, the operation is scheduled for execution. If the second operator takes precedence over the first, or is equal to the first, the first operation is delayed and the second operator is compared against the third.

Example: $(A+B*C\uparrow D)$

The operation $C\uparrow D$ is evaluated first. The multiplication is then performed and finally the addition.

RELATIONAL EXPRESSIONS

The form of a relational expression is:

E1 OP E2

where E1, E2 are arithmetic expressions,

OP is an operator belonging to the set:

<u>Operator</u>	<u>Meaning</u>
=	Equal to
EQ	Equal to
<>	Not equal to
><	Not equal to
NEQ	Not equal to
>	Greater than
GTR	Greater than
>=	Greater than or equal to
GEQ	Greater than or equal to
=>	Greater than or equal to
<	Less than
LSS	Less than
<=	Less than or equal to
=<	Less than or equal to
LEQ	Less than or equal to

Notice that there is more than one form for each relational operator. This is a convenience feature and the choice of form is arbitrary.

A relation is TRUE if E1 and E2 satisfy the relation specified by OP. A relation is FALSE if E1 and E2 do not satisfy the relation specified.

Example: A = 1
D-W NEQ T*4
0.123 = SIN(X)

LOGICAL EXPRESSIONS

The general form of a logical expression is:

R1 OP R2

where R1, R2 are relational expressions and,

OP is one of the Boolean operators AND or OR.

Rules

1. The Boolean operators are defined as follows:
R1 AND R2 true only if R1 and R2 are true,
R1 OR R2 false only if R1 and R2 are false.
2. Precede and follow Boolean operators AND and OR with a relational expression.
3. Do not enclose relational or logical expressions within parentheses.
4. Logical expressions are evaluated from left to right.

Examples: A=1 AND B NEQ 10
T<10 OR T>20
A=B AND A=C OR C=A*A

SIMPLE ASSIGNMENT (LET)

The LET statement is not a statement of algebraic equality. It is an instruction to the computer to do certain computations and to assign the answer to a certain variable. Several variables may be assigned the same value by placing them in a list on the left half. Each LET statement is of the form:

LET variable = expression

or

LET variable list = expression

Examples: 100 LET X=X+1
259 LET W7=(W-X4↑3)*(Z-A/(A-B))-17
605 LET A=B=C=D=1.0
210 LET X=Y=N
391 I=J=K=N+N

The LET is optional.

100 LET A=1 is equivalent to 100 A=1.

GO TO

At times, it may not be desirable to have all statements executed in the order in which they appear in the program. An example of this occurs in the MAXSIN program where the computer has calculated the values X0, M and D and printed them in line 175. We did not want the program to go on to the END statement yet, but we wanted it to go through the same process for a different value of D. So we directed the computer to go back to line 100 with a GO TO statement. Each GO TO statement is of the form:

GO TO line number

Example: 150 GO TO 75

ON - GO TO

The simple GO TO statement provides a single branched switch. The ON--GO TO statement provides a multibranched switch. The form of the statement is:

ON expression GO TO line number,line number,...line number

The expression is any valid BASIC expression, and the line numbers are those to which the statement transfers depending on the value of the expression.

Example: 230 ON X + Y GO TO 275,490,150

This statement is directed to line 575,490 or 150 depending on whether the value of the expression X + Y is 1, 2 or 3.

The value of the expression is truncated to an integer if it is not already an integer. For example, if X + Y equals 2.5, the value is truncated to 2, and the program branches to line 490, the second line number in the list.

IF - THEN/IF - GO TO

At times the normal sequence of statements is to be jumped if a certain relationship holds true. For this we use an IF--THEN statement, sometimes called a conditional GO TO statement. Such a statement occurred at line 130 of MAXSIN. Each IF--THEN statement is of the form:

IF relational expression THEN line number

or

IF relational expression GO TO line number

or

IF logical expression THEN line number

or

IF logical expression GO TO line number

Examples: 340 IF SIN(X)<=M THEN 630
360 IF G=0 THEN 1390
120 IF A NEQ B GOTO 230
214 IF A=1 AND B=2 GOTO 253
934 IF A=1 OR A=3 OR A=5 THEN 500
320 IF I GEQ 10 AND J LSS 20 GOTO 100

The first statement asks whether the sine of X is less than or equal to M, and directs the computer to go to line 630 if it is. The second statement asks if G is equal to zero, and if so directs the computer to line 1390. In each case, if the answer to the question is no, the computer proceeds to the next line of the program. GO TO may be used in place of the THEN statement as shown in the example.

FOR AND NEXT

The FOR and NEXT statements in loops have already been encountered. Both statements must appear in a loop: the FOR at the entrance and the NEXT at the exit (directing the computer back to the entrance). Each FOR statement is of the form:

FOR variable = formula TO formula STEP formula

Most commonly, the formulas are integers and the STEP is omitted, which means that a step size of plus one is assumed. The accompanying NEXT statement is simple in form. However, the variable must be exactly the same as the one following FOR in the FOR statement. The form of the NEXT statement is:

NEXT variable

Examples: 130 FOR X=0 TO 3 STEP 0.5

```
(A)      .
          .
          .
          180 NEXT X

          120 FOR X4=(17+COS(X))/3 TO 3 * SQR(10) STEP 1/4

(B)      .
          .
          .
          235 NEXT X4

          240 FOR X=8 TO 3 STEP -1

(C)      .
          .
          .
          270 NEXT X

          456 FOR J=-3 TO 12 STEP 2

(D)      .
          .
          .
          470 NEXT J
```

Notice that the step size may be a formula (1/4), a negative number (-1), or a positive number (2). In example (B) lines 120 and 235, the successive values of X will be 8, 7, 6, 5, 4, 3. In the last example, on successive trips through the loop J will take on the values -3, -1, 1, 3, 5, 7, 9, and 11.

If the initial, final, or step size values are given as formulas, the formulas are evaluated only once upon entering the FOR statement. The control variable can be changed in the body of the loop. The exit test always uses the latest value of this variable.

If 150 FOR Z=2 TO -2 is written without a negative step size, the loop is not executed, and the computer immediately goes to the statement following the corresponding NEXT statement.

READ AND DATA

A READ statement is used to assign values from a DATA statement to the listed variables. Neither statement may be used without at least one of the other type. A READ statement causes the variables listed in it to be given, (in order) the values of next available numbers in the DATA statements. Before the program is run, the computer groups all of the DATA statements, in the order in which they appear, into a large data block. Each time a READ statement is encountered anywhere in the program, the data block supplies the next available number or numbers. If the data block runs out of data, with a READ statement still asking for more, the program is assumed to be finished.

Since data must be read in before the program can be worked with, normally READ statements are placed near the beginning of a program. The location of DATA statements is unimportant, as long as they are in the correct order. A common practice is to put all DATA statements together immediately before the END statement.

Each READ statement is of the form:

READ sequence of variables

and each DATA statement is of the form:

DATA sequence of numbers

Examples: 150 READ X,Y,Z,X1,Y2,Q9
330 DATA 4,2,1.7
340 DATA 6.734E-3,-174.321,3.14159265

234 READ B(K)
263 DATA 2,35,7,9,11,10,8,6,4

100 READ R(I,J)
440 DATA -3,5,-9,2,37,2.9876,-437.234E-5
450 DATA 2.765,5.5576,2.3678E2

Remember that only numbers are put in DATA statements, formulas are not allowed.

INPUT

At times it is desirable to have data entered while the program is running. This is particularly true when one person writes the program and stores it in the computer's memory, and data is to be supplied by others. This may be accomplished by using an INPUT statement, which is similar to a READ statement, but does not draw numbers from a DATA statement. Each INPUT statement is of the form:

INPUT sequence of variables

If, for example the user is to supply values for X and Y in a program, he writes:

```
140 INPUT X,Y
```

before the first statement that is to use either of the two values. When the INPUT statement is encountered, the computer types a question mark on the printout and waits for input. The user types two numbers, separated by a comma, presses the return key, and the computer continues with the remainder of the program.

Frequently, an INPUT statement is accompanied by an explanatory PRINT statement. If in a program:

```
120 PRINT "YOUR VALUES OF X, Y, AND Z ARE";  
130 INPUT X,Y,Z
```

the computer types out:

```
YOUR VALUES OF X, Y AND Z ARE?
```

Without the semicolon at the end of line 120, the question mark would have been printed on the next line.

Data entered by an INPUT statement is not saved by the program.

Also, it may take a long time to enter a large amount of data using INPUT. This statement should therefore be used only when small amounts of data are to be entered or when it is necessary to enter data during the program run, as it is with game playing programs.

PRINT

The PRINT statement has a number of different uses. It is discussed in more detail in Chapter 3. The most frequent uses are:

- (A) To print out the result of some computations,
- (B) To print out verbatim a message included in the program,
- (C) A combination of A and B.

Up to now only examples of A and B have been seen in the sample programs. Each type is slightly different in form, but all start with PRINT after the line number.

Examples of Type A:

```
100 PRINT X,SQR(X)
135 PRINT X,Y,Z,B*B-4*A*C,EXP(A-B)
```

Statement 100 prints the value of X and then, a few spaces to the right, its square root. Statement 135 prints five different numbers: X,Y,Z, B^2-4AC , and $e^{(A-B)}$. The computer calculates the two formulas and prints the values, if values have already been assigned to A, B, and C. It can print up to five numbers per line in this format.

Examples of Type B:

```
100 PRINT "NO UNIQUE SOLUTION"
430 PRINT "X VALUE","SINE","RESOLUTION"
```

Both examples have been seen in the sample programs. The first prints the statement:

```
NO UNIQUE SOLUTION
```

The second prints the three labels with spaces between them. The labels in statement 430 automatically align with three numbers called for in a PRINT statement, as seen in the program MAXSIN.

Examples of Type C:

```
150 PRINT "THE VALUE OF X IS"; X
315 PRINT "THE SQUARE ROOT OF";X;"IS";SQR(X)
```

If statement 150 has computed the value of X as 3, it prints:

```
THE VALUE OF X IS 3
```

If statement 315 has computed the value of X as 625, it prints:

```
THE SQUARE ROOT OF 625 IS 25
```

In statements of type C, the semicolon is used to minimize space.

DIM

To enter a list or table with a subscript greater than 10, a DIM statement must be used to direct the computer to save enough room for the list or table.

Examples: 120 DIM H(35)
135 DIM Q(5,25)

The first statement enables a list of 35 items to be entered and the second, a table 5 by 25.

FUNCTIONS

This section describes three more BASIC functions and a statement that allows a user to define his own functions.

INT

INT frequently appears in algebraic computations as [X], and gives the greatest integer not greater than X. Thus INT(2.35) equals 2, INT(-2.35) equals -3, and INT(12) equals 12.

One use of the INT function is to round numbers. For example to round 2.9, to the nearest integer using the function INT(X+.5):

```
INT(2.9 + .5)=INT(3.4)=3
```


The function $\text{INT}(X+.5)$ will round the number that is midway between two integers up to the larger of the integers.

It can also be used to round to any specific number of decimal places. For example, $\text{INT}(10*X+.5)/10$ rounds X correct to one decimal place; $\text{INT}(100*X+.5)/100$ rounds X correct to two decimal places; and $\text{INT}(X*10^D+.5)/10^D$ rounds X correct to D decimal places.

RND

The function RND is a pseudo random number generator. It requires a single argument, which has the following meanings:

If the argument is positive, it is used to initiate the random number sequence.

If the argument is negative, a random number is used to initiate the random number sequence.

If the argument is zero, RND supplies a random number. The first use of $\text{RND}(0)$ in a program always yields the same random number.

A positive or negative argument will normally be used to initiate a sequence of random numbers, after which a zero argument will be used repeatedly.

If the initial value used for the argument is 2 or any power of 2, the same initial random number results.

If the first twenty random numbers are desired, the following program may be used to get twenty-six digit decimal.

```
Example: 100 X=RND(1)
          110 FOR I=1 TO 5
          120 PRINT RND(0),RND(0),RND(0),RND(0)
          130 NEXT I
          1000 END
          ?RUN
```

2.20857E-02	0.38263	0.98879	0.10399
0.68124	3.68996E-02	0.30806	0.60544
0.94384	0.54407	6.34831E-02	0.72934
0.99684	0.12717	0.86718	0.89592
0.94197	0.20996	0.99979	1.77987E-0

If, on the other hand, twenty random one-digit integers are desired, we can change line 120 to read:

```
120 PRINT INT(10*RND(0));
?RUN
```

to obtain:

0	3	9	1
6	0	3	6
9	5	0	7
9	1	8	8
9	2	9	0

The kind of random numbers may also be varied. For example, to obtain twenty random numbers ranging from 1 to 9 inclusive change line 120 to read:

```
120 PRINT INT(9*RND(0)+1);
?RUN
```

1	4	9	1
7	1	3	6
9	5	1	7
9	2	8	9
9	2	9	1

or to obtain random numbers which are integers from 5 to 24 inclusive change line 120 to read:

```
120 PRINT INT(20*RND(0)+5);
?RUN

5 12 24 7 18 5 11 17 23 15 6 19 24 7
22 23 9 24 5
```

In general, if random numbers are to be chosen from A integers of which B is the smallest, call for:

```
INT(A * RND(0) + B)
```

after first having initiated the random number sequence with a positive or negative argument, as in line 100 of our sample program.

If you were to run the first version of our sample program again, the same twenty numbers could be generated in order. However a different set may be generated by discarding some of the random numbers or by starting the series using a negative argument. The following program finds and stores the first ten random numbers.

It then finds and prints the next twenty random numbers. By comparing this with the earlier program notice, the first ten random numbers are the same as the second ten in the first program.

Example: 100 LET Z=RND(1)
 110 FOR I=1 TO 10
 120 LET Y=RND(0)
 130 NEXT I
 140 FOR I=1 TO 5
 150 PRINT RND(0),RND(0),RND(0),RND(0)
 160 NEXT I
 1000 END
 ?RUN

6.34831E-02	0.72934	0.99684	0.12717
0.86718	0.89592	0.94197	0.20996
0.99979	1.77987E-02	0.82975	0.32772
0.95100	0.54719	0.53430	0.68283
0.38419	0.34709	0.10066	0.49632

SGN

The function SGN allows the user to test for the sign of any value. The form is SGN(argument) and it yields +1, -1, or 0 depending on the value of the argument. The options are:

<u>Argument Value</u>	<u>Yields</u>
Zero	0
Positive, not zero	+1
Negative, not zero	-1

Examples: SGN(0) yields 0
 SGN(-1.82) yields -1
 SGN(989) yields +1
 SGN(-.001) yields -1
 SGN(-0) yields 0

DEF

In addition to making use of the standard functions, any other function that is to be used repeatedly in a program may be defined. A DEF statement is used to define such a function. The name of the defined function must be three letters, the first two of which are FN. Hence up to 26 functions may be defined in one program: FNA, FNB, FNC, and so on.

The usefulness of DEF can best be illustrated in a program. For example, where the function $E-(X^2)$ is frequently needed. Introduce the function by the line:

```
130 DEF FNE(X) = EXP(-X^2)
```

and later on call for various values of the function by FNE(.1), FNE(3.45), FNE(A+2), etcetera. DEF can be a great time-saver when values of a function for a number of different values of the variable are needed.

The DEF statement may be inserted anywhere in the program, and the expression to the right of the equal sign may be any formula that can be accommodated on one line. It may include functions defined by other DEF statements, and it can involve variables other than the one denoting the argument of the function. For example, if FNR is defined by:

```
170 DEF FNR(X)=SQR(2+LOG(X)-EXP(Y*Z)*(X+SIN(2*Z)))
```

and if values have been previously assigned to Y and Z, then FNR(2,175) may be used. Before the next use of FNR, new values may be assigned to Y and Z.

DEF is generally limited to cases where the value of the function can be computed within a single BASIC statement. Often more complicated functions, or an entire program section, must be calculated at different points within a program. For these functions, the GOSUB statement will frequently be useful. It is described in the following section.

END

Every program must have an END statement, and it must be the statement with the highest line number in the program. Its form is:

```
line number END
```

Example: 999 END

CHAPTER 3 - ADDITIONAL BASIC STATEMENTS

Several types of BASIC statements were not covered in Chapters 1 or 2. They are:

GOSUB and RETURN
STOP
REM
RESTORE

GOSUB AND RETURN

When a particular part of a program is to be used more than once, or possibly at several different places in the program, it is most efficiently programmed as a subroutine. The subroutine is entered with a GOSUB statement.

Example: 190 GOSUB 310

The line number, 310, is the first statement of the subroutine.

The last line of the subroutine should be a RETURN statement, directing the computer to return to the earlier part of the program.

Example: 450 RETURN

If statement 450 is the last line in the subroutine entered in the previous example, it directs the computer to return to the first line numbered greater than 190 and to continue the program from there.

A GOSUB statement may be used inside a subroutine to execute yet another subroutine. This is called nested GOSUBs. Note: the last statement of the subroutine must be a RETURN. This command directs the computer to return to the main program after it has finished the computations within the subroutine. The subroutine must therefore, contain at least one RETURN statement.

A subroutine should never contain a GOSUB statement that refers to one of the subroutines already entered. Recursion is not allowed.

The following example, a program for determining the greatest common divisor of three integers using the Euclidean algorithm, illustrates the use of a subroutine.

```
Example: 100 PRINT "A","B","C","GCD"
          110 READ A,B,C
          120 LET X=A
          130 LET Y=B
          140 GOSUB 230
          150 LET X=G
          160 LET Y=C
          170 GOSUB 230
          180 PRINT A,B,C,G
          190 GO TO 110
          200 DATA 60,90,120
          210 DATA 38456,64872,98765
          220 DATA 32,384,72
          230 LET Q=INT(X/Y)
          240 LET R=X-Q*Y
          250 IF R=0 THEN 290
          260 LET X=Y
          270 LET Y=R
          280 GO TO 230
          290 LET G=Y
          300 RETURN
          999 END
?RUN
```

A	B	C	GCD
60	90	120	30
38456	64872	98765	1
32	384	72	8

The first two numbers are selected in lines 120 and 130, and their GCD is determined in the subroutine, lines 230 - 300.

The GCD just calculated is called X in line 150, the third number is called Y in line 160, and the subroutine is entered from line 170 to find the GCD of these two numbers. This number, and the GCD of the three given numbers, is printed out with the three numbers in line 180.

STOP

The STOP statement is equivalent to GO TO XXXXX, where XXXXX is the line number of the END statement in the program. It is useful in programs having more than one natural finishing point. For example, the following two program portions are exactly equivalent.

Example:	250 GO TO 999	250 STOP
	.	.
	.	.
	.	.
	340 GO TO 999	340 STOP
	.	.
	.	.
	.	.
	999 END	999 END

REM

The REM statement allows the insertion of explanatory remarks in a program. The form is:

REM any comment.

The computer completely ignores the part of the line following REM, allowing comments to be included for using the program, for identifying of the parts of a program, or for any further explanations. Although anything following REM is ignored, the line number of a REM statement may be used in a GOSUB or an IF--THEN statement.

Example:

```
100 REM INSERT DATA IN LINES 900-998.  THE FIRST
110 REM NUMBER IS N, THE NUMBER OF POINTS.THEN
120 REM THE DATA POINTS THEMSELVES ARE ENTERED, BY

200 REM THIS IS A SUBROUTINE FOR SOLVING EQUATIONS.
    .
    .
    .
300 RETURN
    .
    .
    .
520 GOSUB 200
```


RESTORE

Sometimes it is necessary to use data in a program more than once. The RESTORE statement permits the reading of data as many additional times as necessary. Whenever RESTORE is encountered in a program, the computer restores the data block pointer to the first item of data. A subsequent READ statement then starts reading the data again.

Warning: If the data items to be reused are preceded by code numbers or parameters, superfluous READ statements must be inserted to bypass the numbers.

For example, the following section of a program reads the data, restores the data block to its original state, and rereads the data. Note the use of line 570 to bypass the value of N, which is already known.

```
Example: 100 READ N
          110 FOR I=1 TO N
          120 READ X
            .
            .
            .
          200 NEXT I
            .
            .
            .
          560 RESTORE
          570 READ X
          580 FOR I=1 TO N
          590 READ X
```

CHAPTER 4 - ADVANCED BASIC

Chapter 2 discussed how to write programs in BASIC. Chapters 4 to 6 now examine some capabilities of BASIC that were not previously mentioned. These include:

- Alphanumeric data and string manipulation
- Files
- Matrices

ALPHANUMERIC DATA AND STRING MANIPULATION

Alphanumeric data, names, and other identifying information can be handled in BASIC using string variables. It is possible to enter, store, compare, and print out alphanumeric and certain special characters in the BASIC character set.

A STRING is any sequence of alphanumeric and certain special characters in the basic character set not used for control purposes.

A STRING VARIABLE is denoted by a letter followed by a dollar sign. For example, A\$, B\$, and X\$ denote string variables. There is no limit to the length of a string variable although it must be less than 72 characters because BASIC statement (including LET) must not exceed one line.

DIM

Strings can be set up as one-dimensional arrays only. If a two-dimensional array is requested, the error message DIMENSION ERROR is received.

```
Examples: 100 DIM A(5),C$(20),A$(12),D(10,5)
          200 DIM R$(35)
          300 DIM M$(15),B$(15)
```

In line 100, only C\$ and A\$ are string variables. R\$, as dimensioned in line 200, allocates storage space for 35 entries.

LET

Strings and string variables may appear in only two forms of the LET statement. The first is used to replace a string variable with the contents of another string variable.

Example: 156 LET G\$=H\$

The second is used to assign a string to a string variable.

Example: 160 LET J\$="THIS STRING"

Line 160 assigns the string THIS STRING to variables A\$, G\$ and J\$. Any valid expression or string may be used.

READ AND DATA

READ statements can contain string variables intermixed with ordinary variables. In the corresponding DATA statements, every item corresponding to a string variable in the READ statement must be a valid string. If the string contains any characters that have special meaning in BASIC - such as commas, semicolons, leading or trailing spaces - they must be enclosed in quotation marks. Unquoted strings must begin with an alphabetic character.

```
Example: 100 READ A$,B$,C$,D$,A,E$
          200 DATA THE,"","PEOPLE","YES--","500",OF THEM.
          300 PRINT A$;B$;C$;D$;A;E$
          999 END
```

This program prints THE PEOPLE, YES--500 OF THEM. The DATA statement has quotation marks around B\$ because it is a blank space, and around the value for C\$ because it includes a comma.

INPUT

INPUT statements may also contain string variables intermixed with ordinary variables. Every item corresponding to a string variable in the INPUT statement must be a valid string variable. If the string contains characters that have special meaning in

BASIC, it must be enclosed by quotation marks. If the string begins with anything other than an alphabetic character it must be enclosed in quotation marks.

Example: 110 INPUT L\$(17),M\$,N\$(I)

PRINT

The PRINT statement may also contain string variables intermixed with ordinary variables. When a string variable is encountered that has not been assigned, the PRINT statement ignores that variable. A semicolon after a string variable in a PRINT statement causes the printout of the variable following that string to be directly connected to the string variable.

Example: 135 PRINT A,16,B\$,C\$;N
140 PRINT 100+I,"DATA",L\$;M\$;N\$
150 PRINT S\$

IF - THEN/IF - GOTO

Only one string variable is allowed on each side of the IF-THEN relation sign. The six standard relations are valid (=, <>, <, >, <=, and >=). When strings of different lengths are compared, the shorter string and the corresponding part of the longer string are used. If they compare, the shorter string is taken to be the lesser of the two.

Example: 100 IF N\$="SMITH" THEN 105
200 IF A\$<>B\$ GOTO 205
300 IF "JUNE" <=M\$ GO TO 305
400 IF D\$>="FRIDAY" THEN 600

Quotation marks must be used around the string to be compared, as above, unless it is referenced in the IF--THEN statement by a string variable name.

Characters are compared in their BASIC code representations. The collating sequence used in comparing is listed in Appendix D.

MORE ABOUT PRINTING

Although the format of the printout is automatically supplied for the beginner, the PRINT statement, the TAB function, and image formatted output permit a greater flexibility for the advanced programmer in establishing different formats for his output.

PRINT

The teletypewriter line is divided into five zones of fifteen spaces each. Some control of the use of these zones comes from the use of the comma. A comma is a signal to move to the next print zone or, if the fifth print zone has been used, to move to the first print zone on the next line.

Shorter zones can be made by using the semicolon. The zones are two characters longer than the length of the number to be printed. An additional space is reserved for the sign. Thus a 2 digit number will have a five character zone and an 11 digit number will have a 14 character zone. A semicolon is a signal to move to the next short print zone or, if the last such zone on the line has been used, to move to the first print zone of the next line.

The first space in any print zone is reserved for the sign, even though it is not printed if it is plus.

If the following program is typed:

```
100 FOR I=1 TO 15
110 PRINT I
120 NEXT I
999 END
```

The teletypewriter prints 1 at the beginning of the first line, 2 at the beginning of the second line, and so on, finally printing 15 at the beginning of the fifteenth line. But if line 110 is changed to read:

```
110 PRINT I,
```

The numbers are printed in zones:

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

Using a semicolon in place of the comma produces:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Remember that a label inside quotation marks is printed exactly, as it appears, and also that the end of a PRINT statement signals a new line, unless a comma or semicolon is the last symbol. The instruction:

```
150 PRINT X,Y
```

results in the printing of two numbers and the return to the next line, but

```
150 PRINT X, Y,
```

results in the printing of two numbers and no return. The next number is printed in the third zone on the same line as the values of X and Y.

Since the end of a PRINT statement signals a new line, a statement such as

```
250 PRINT
```

causes the completion of a partially filled line, as illustrated in the following section of a program:

```
100 FOR M=1 TO N
110 FOR J=1 TO M
120 PRINT B(M,J);
130 NEXT J
140 PRINT
150 NEXT M
```

The program prints B(1,1) and next to it B(1,2). Without line 140, the teletypewriter then continues printing B(2,1), B(2,2), B(3,1), B(3,2), if there is sufficient space remaining on the line. Line 140 directs the teletypewriter to start a new line after printing the B(1,1) value corresponding to M=1, the B(2,2) value corresponding to M=2, and so on.

TAB

The print function TAB permits tabbing of the input terminal whenever the TAB function is used in the PRINT statement, it causes a line pointer to move over to the position indicated by the argument of TAB. The next value to be stored in the print line goes into this position.

Example: 150 PRINT X; TAB(10); Y; TAB(2*N);Z

The argument of TAB refers to a print position on the line. The positions are assumed to run from 0 through 74. In the above example, if the value of N is 10, the pointer moves to the 10th print position after printing the value of X, and to the 20th print position after printing the value of Y.

When utilizing the TAB function, the semicolon should be used in the PRINT statement to minimize field width.

If the argument of TAB is less than the current print position, it is ignored.

All arguments of TAB are treated modulo 75.

RULES FOR PRINTING NUMBERS

The following rules for the printing of numbers will held in interpreting printed results.

If a number is an integer, the decimal point is not printed. If the integer is larger than or equal to 2^{29} (i.e. 536,870,912), the teletypewriter prints the first digit, followed by (1) a decimal point, (2) the next five digits, and (3) an E followed by the appropriate exponent integer. For example, it prints 32,437,580,259 as 3.24376E+10.

For any decimal number, no more than six significant digits are printed.

For a number less than 0.1, the E notation is used unless the entire significant part of the number can be printed as a six decimal number. For example, .03456 means that the number is exactly .0345600, but 3.45600E-2 means that the number has been rounded to .0345600.

PRINT USING AND IMAGE STATEMENTS

Formatted line output may be established by use of the PRINT USING and image statements.

The form of the PRINT USING statement is:

PRINT USING line number,output list

The form of the image statement is:

line number:line image

where	line number	is that of the image statement in the program,
	line image	consists of format control characters and printable constants.

Format control characters are:

- ' (Apostrophe) a one-character field that is filled with the first character in an alphanumeric string regardless of the string length.
- " (Quotation Marks) the replacement field of an alphanumeric string of two or more characters; the field width includes the quotation marks as well as the characters (if any) contained within the marks.
- # (Pound Sign) the replacement field for a numeric character.
- ↑↑↑↑ (Four Up-arrows) indicates scientific notation for a numeric field.

All other characters are treated as printable constants.

The following simple example is part of a program, showing the use of the PRINT USING statement, the line image statement, and format control characters.

```
Example: 100 PRINT USING 120,A$,"$",A,324,X
          120:"      "      '###.##      #####      ##.##↑↑↑↑
```

If the values of A\$, A, and X are FIRST, 12.9, and 24687, then the output is:

```
          FIRST      $ 12.90      324      2.47E+04
```

An image statement must begin with a colon. It is composed of fields which form the print line. There are five types of fields:

- Integer fields
- Decimal fields
- Exponential fields
- Alphanumeric fields
- Literal fields

INTEGER FIELDS

The following rules apply to integer fields.

An integer field is composed of pound signs (#).

Numbers in an integer field are right justified and truncated if they are not integral.

The field is widened to the right if the number is too big.

The field must reserve a place for the algebraic sign.

If the number is greater than 1,073,741,823 in absolute value, asterisks are printed.

```
Example: 100:  ####      #####      ###
          110 READ A,B,C
          120 PRINT USING 100,A,B,C
          130 GO TO 110
          140 DATA 123.45,-34.856,45.7,457.34,-17,89.999
          999 END
          ?RUN
```

```
          123      -34      45
          457      -17      89
```

```
          OUT OF DATA      110
```

DECIMAL FIELDS

The following rules apply to decimal fields:

A decimal field is a string of pound signs (#) with an imbedded period. Note that .## is not a decimal field because the period is not imbedded.

The number is rounded to the number of places specified by the pound signs following the decimal.

The number is right justified, placing the decimal as given in the field definition.

The field will be widened to the right if the number is too large.

The field must include a place for the algebraic sign.

```
Example: 100:  ####.##      ####.####      #####.      #.###
          110 READ A,B,C,D
          120 PRINT USING 100,A,B,C
          130 GO TO 110
          140 DATA 123.456,-34.856,47.7,-.0177
          150 DATA 1.999,876.55,-17,.893
          999 END
          ?RUN

          123.46   -34.8560    48.   -.018
           2.00    876.5500   -17.    .893

          OUT OF DATA   110
```

EXPONENTIAL FIELDS

The following rules apply to exponential fields:

An exponential field is a decimal field followed by four up-arrows (↑↑↑↑), which reserve a place for the exponent.

The pound signs preceding the decimal represent the factor by which the exponent is adjusted.

The number is rounded as with decimal fields.

A place must be reserved for the sign.

```
Example: 100:  #.#####↑↑↑↑      ##.#####↑↑↑↑      ###.↑↑↑↑      #.##↑↑↑↑
          110 READ A,B,C,D
          120 PRINT USING 100,A,B,C,D
          130 GO TO 110
          140 DATA 123.456,-34.856,47.7,-.0177
          150 DATA 1.999,876.55,-17,.893
          999 END
          ?RUN
```

```

.12346E+03    -3.486E+01    48.E+00    -.18E-01
.19990E+01    8.766E+02    -17.E+00    .89E+00

```

```
OUT OF DATA    110
```

ALPHANUMERIC FIELDS

The following rules apply to alphanumeric fields:

The apostrophe is used to print the first character from a string variable or quoted constant.

A field bounded by quotation marks is used to print two or more characters.

In an alphanumeric field of two or more characters, the string is left justified within the field and blank filled or truncated on the right.

```

Example: 100:    "23456"    "THE NAME GOES HERE"    '    ''
110 READ A$,B$,C$,D$,E$
120 PRINT USING 100,A$,B$,C$,D$,E$
130 DATA ABCDEFGHI
140 DATA ABCDEF
150 DATA ABC
160 DATA ABC
170 DATA ABC
999 END
?RUN

```

```

      ABCDEFG    ABCDEF          A    AA

```

LITERAL FIELDS

A literal field is composed of characters or character strings that are not control characters. It appears on the print line exactly as it appears in the image.

```

Example: 100:    THE VALUE FOR A IS    '####.##
110 LET A=100.54
120 LET A$="$"
130 PRINT USING 100,A$,A
999 END
?RUN

```

```

      THE VALUE FOR A IS    $ 100.54

```

GENERAL RULES

The following rules apply in general to formatted line output.

The list elements in the PRINT USING statement may be expressions, variables, numeric constants, and quoted literals.

Numeric list elements must replace numeric fields, and alphanumeric elements must replace alphanumeric fields, or the error message BAD IMAGE XXX is received.

If the output list contains more elements than there are replaceable fields in the image statement, a carriage return is supplied after the last field in the image, and the image is reused. The extra elements are printed on a second line only if they match the image fields that are to be used.

```
Example: 100 PRINT USING 120
          110 PRINT
          120:          I          I↑2          I↑3
          130:  #####          #####          #####
          140 FOR I=1 TO 6
          150 LET A(I)=I
          160 LET B(I)=I↑2
          170 LET C(I)=I↑3
          175 NEXT I
          180 FOR I=1 TO 6 STEP 2
          190 PRINT USING 130,A(I),B(I),C(I),A(I+1),B(I+1),C(I+1)
          200 NEXT I
          999 END
          ?RUN
```

I	I↑2	I↑3
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216

The following program demonstrates one kind of application in which the formatted output line is useful.

```
Example: 100 PRINT USING 170
          110 PRINT
          120 FOR I=1 TO 4
          130 READ A$,A,B
          140 LET T=A*B
          150 PRINT USING 180,A$,A,B,"$",T
          160 NEXT I
```

```

170:NAME                HRS WORKED                RATE                PAY
180:"                  "                ####.##          ##.###/HR      '####.##
190 DATA ANDREWS,47.5,3.987,KELLY,40,2.865,MANLEY,46,3.020
200 DATA ZUMPANO,42.34,4.255
999 END
?RUN

```

NAME	HRS WORKED	RATE	PAY
ANDREWS	47.50	3.987/HR	\$ 189.38
KELLY	40.00	2.865/HR	\$ 114.60
MANLEY	46.00	3.020/HR	\$ 138.92
ZUMPANO	42.34	4.255/HR	\$ 180.16

CHAPTER 5 - FILE INPUT/OUTPUT

DATA FILES

BASIC data files are standard OS-3 files. BASIC will accept EDIT, COSY, or BCD files as input. Data files can be created by a BASIC program or the OS-3 text editor.

Files must be opened before the program can read or write on them. An open file is always in either READ or WRITE mode. The SCRATCH statement establishes a file in write mode. The RESTORE statement establishes it as read mode.

FILE OPENING

The form of the open statement is

```
OPEN %I,"fname"
```

where I file designator; a number (logical unit)
 between 1 and 20,
 "fname" any valid TCL/EDIT filename of 1 to 8 characters.

A BASIC program references files by specifying a logical unit (LUN). The OPEN statement, given a filename of 1-8 characters, prepares the file for data transmission by assigning a logical unit number to the file. The logical unit number should be used for any subsequent references to this open file from within the program.

```
Example: 100 OPEN %1,"FILE1"  
         200 OPEN %2,"FILE2"  
         300 OPEN %3,"INPDAT"
```

An OPEN statement sets an existing file in read mode. If it references a file that does not yet exist, then a file with the specified name is created and put into write mode.

FILE READING

The file READ statement takes information from an open file and assigns the values to variables specified in the list given with the READ statement. A file must be in read mode before a READ is allowed. The format of the file READ statement is:

```
READ %logical unit,list
```

Example: 100 OPEN %1,"DATAFILE"
150 FOR I=1 TO 10
200 READ %1,X(I),Y(I),Z(I)
250 NEXT I

FILE WRITING

A file WRITE statement inserts information into a data file. It operates similar to a PRINT statement with a list of values to transfer, except that when a print line is finished the line is put into the next sequential position of the data file specified in the WRITE statement. A file specified in a WRITE statement must be in write mode. Refer to the descriptions of the SCRATCH and OPEN statements. After a WRITE the file position is automatically incremented to the next line. The format of the WRITE statement is:

```
WRITE %logical unit,list
```

Example: 100 OPEN %5,"DATFILE"
200 SCRATCH %5
300 FOR I=1 TO 100
400 WRITE %5,I,I*I,I*I*I
500 NEXT I
600 CLOSE %5

END-OF-FILE CHECK

This statement is of the form:

```
IF END %logical unit THEN line number
```

If the last READ statement detected an end-of-file, the program goes to the line number specified in the THEN clause. Otherwise it executes the next statement.

If the program continues reading a file after encountering an end-of-file the diagnostic UNCHECKED END-OF-FILE XXX is printed and the program stops.

FILE RESTORING

The RESTORE statement is of the form:

```
RESTORE %logical unit
```

This statement has two functions. It positions the file at the beginning of its data and sets the file in read mode. A file which is being written cannot be read until it is restored.

FILE SCRATCHING

The SCRATCH statement is of the form:

```
SCRATCH %logical unit
```

This statement causes a currently open file to be set into write mode. It also deletes any data on a currently existing file. After scratching the contents of a file, a subsequent WRITE starts at the beginning of the file. This means a file cannot be modified once written except by being copied.

FILE CLOSING

The CLOSE statement is of the form:

```
CLOSE %logical unit
```

CLOSE makes the file unavailable for reading and writing from the program. It also clears the output buffer for a write mode file.

All output files MUST be closed or some of the data written may be lost. When a program reaches the END statement any open files are automatically closed by BASIC.

CHARACTERISTICS OF BASIC DATA FILES

BASIC creates BCD (card images) format files when using the WRITE or PRINT on data files. Interactive BASIC can READ all standard OS-3 file types. Files created with a WRITE have the data items put sequentially on the file with items in the same line separated by commas. The file READ statement expects data to be separated by commas.

To prepare a data file outside of BASIC, type in the data items separated by commas. String constants may appear at any point. The rules for values in a file are the same as for the DATA statement list entries.

Example: SALT,PEPPER,SUGAR,NUTMEG,COFFEE,MARCH
 -1,0
 -1,0,0

The PRINT statement may be used on files and each line of the file will be the same as a normal PRINT on the user's terminal.

CHAPTER 6 - MATRICES

The matrix operation statements available in BASIC are among the most powerful and useful in the entire language.

Following is a list of matrix statements.

MAT READ A,B,C	Read matrices A, B, and C, the dimensions of which have been previously specified. Data is read in row-wise sequence.
MAT PRINT A,B;C	Print matrices A, B, C and store A and C in the regular format, but store B closely packed.
MAT C=A+B	Add matrices A, and B and store the result in matrix C.
MAT C=A-B	Subtract matrix B from matrix A and store the result in matrix C.
MAT C=A*B	Multiply matrix A by B and store the result in matrix C.
MAT C=INV(A)	Invert matrix A and store the result in matrix C.
MAT C=TRN(A)	Transpose matrix A and store the result in matrix C.
MAT C=(K)*A	Multiply matrix A by the value represented by K. K may be either a number or an expression, but in either case it must be enclosed in parentheses.
MAT C=CON	Set each element of matrix C to one. CON means constant.
MAT C=ZER	Set each element of matrix C to zero.
MAT C=IDN	Set the diagonal element of matrix C to ones and the non-diagonal elements to zeroes, yielding an identity matrix.

MAT READ AND MAT PRINT

Using the MAT READ and MAT PRINT statements, data may be read into or printed from a matrix without having to reference each element of the matrix individually.

```
Example: 100 MAT READ A,F,H,G
          150 MAT PRINT C
          175 MAT READ Z
          190 MAT PRINT A,L
```

Information is read into a matrix using the DATA statement. The elements in the DATA statement are taken in row order, that is,

$$A_{1,1}, A_{1,2}, \dots, A_{1,n}, A_{2,1}, A_{2,2}, \dots, A_{m,n}$$

Information is read from the DATA statements until the matrix array is completely filled. Partial matrices cannot be read or printed.

```
Example: 110 DIM L(2,3),M(2,2)
          150 MAT READ L,M
          160 LET L(2,2)=-2*L(2,2)
          200 MAT PRINT L,M
          500 DATA 1,2,3,4,5,6,3,-12,0,7
```

Line 110 defines L as a 2 by 3 matrix and M as a 2 by 2 matrix. The MAT READ statement reads in row order from the DATA statement at line 500. The matrix element L is recomputed at line 160. The two matrices are then printed to yield:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & -10 & 6 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 3 & -12 \\ 0 & 7 \end{bmatrix}$$

MATRIX ARITHMETIC

It is possible to add, subtract, and multiply matrices using the matrix arithmetic statements. The matrix dimensions must be conformable for each operation. If dimensions are not conformable, execution is stopped and a dimension error message is received.

Matrix arithmetic statements may take the forms:

MAT C=A+B MAT C=A-B MAT P=Q*R

Only one operation can be performed in each statement.

Example: CALCULATE H=A*B+E-K

612 MAT H=A*B
615 MAT H=H+E
618 MAT H=H-K

SCALAR MULTIPLICATION

A matrix can be multiplied by a scalar expression using a statement of the form:

MAT X=(expression)*D

where X and D are matrices and the expression in parentheses is a scalar quantity. The parentheses are required to indicate scalar rather than matrix multiplication. Only one operation per statement is allowed:

Example: 100 MAT F=(2)*G
150 MAT Q=(2.33+M)*Q
750 MAT B=(N)*A

IDENTITY MATRIX

An identity matrix is defined by a statement of the form:

MAT B=IDN or MAT R=IDN(expression,expression)

In the first statement, matrix B is established as an identity matrix. If B is not defined to be square, a dimension error message is received. In the second statement, the size of the identity matrix R is determined at execution time by the value of the expression enclosed in parentheses.

Examples: 190 MAT A=IDN
100 MAT V=IDN(2*N+1,2*N+10)
120 MAT B=IDN(Q,Q)
130 MAT W=IDN
140 MAT C=IDN(1,1)

MATRIX TRANSPOSITION

Matrices are transposed using the form:

```
MAT Y=TRN(Z)
```

where Y and Z are both matrices. The transpose of matrix A replaces matrix Y. Y and Z must conform. Matrix transposition in place (MAT A=TRN(A)) is not allowed.

Examples: 300 MAT G=TRN(H)
400 MAT U=TRN(V)

MATRIX INVERSION

Matrices are inverted using the form:

```
MAT I=INV(J)
```

where I and J are both matrices. I contains the inverse of J. I and J must conform. Matrix inversion in place (MAT A=INV(A)) is not allowed. If a matrix is singular, the message NEARLY SINGULAR MATRIX XXX is received.

Examples: 500 MAT K=INV(L)
560 MAT A=INV(B)

MATRIX ZER AND CON FUNCTIONS

The ZER function is used to zero out all elements of a matrix. It may also be used to redefine the dimensions of a matrix during execution as described in "Dimensioning". For example:

```
MAT C=ZER
```

zeroes out the elements of matrix C.

The CON function is used to set all elements of a matrix to ones. For example:

```
MAT C=CON
```

sets all elements of matrix C to ones.

DIMENSIONING

Every matrix variable used in a program must be given a single-letter name. A matrix variable must be defined in a DIM statement, which reserves the amount of storage required by the matrix variable during execution of the program. For example:

```
DIM P(3,4),Q(5,5)
```

This DIM statement defines two matrices, P and Q. P is defined as a 12 element matrix, and Q as a 25 element matrix. Note that the first element of P is P(1,1) and the last element P(3,4). The elements of Q run from Q(1,1) through Q(5,5). All matrix variables must be doubly dimensioned, as shown here.

Before any computation using the MAT statements is performed, the precise dimensions of all matrices to be used in the computation must be declared. Four MAT statements are used for this purpose:

```
MAT READ C(M,N)
MAT C=ZER(M,N)
MAT C=CON(M,N)
MAT C=IDN(N,N)
```

The first three statements specify matrix C as consisting of M rows and N columns. The fourth statement specifies matrix C as a square matrix of N rows and N columns.

These same statements may be used to redimension a matrix during a run. A matrix may be redimensioned to either a larger or a smaller matrix, provided the new dimensions do not require more storage space than was originally reserved by the DIM statement. To illustrate, consider the following.

```

Example: 110 DIM A(8,8),B(8,8),C(8,8)
          150 MAT READ A(2,2),B(2,2)
          160 MAT C=ZER(2,2)
          .
          .
          .
          200 MAT A=IDN(8,8)
          210 MAT READ B(4,4),C(4,4)

```

Note that the DIM statement reserves enough storage to accommodate three matrices, each consisting of 64 elements. The initial MAT READ specifies the dimensions of both matrices A and B as 2 rows and 2 columns.

The MAT READ also reads the number of values required by the dimensions into the storage that was reserved by the DIM statement. It reads them in row-wise sequence. In the initial MAT READ, the elements are read in the order A(1,1), A(1,2), A(2,1), A(2,2), B(1,1), B(1,2), B(2,1), and B(2,2). Statement 160 uses the ZER to specify dimensions and to zero the elements of matrix C. Statements 200 and 210 illustrate redimensioning. Matrix A is redimensioned as an 8 row, 8 column identity matrix; and matrices B and C are redimensioned as 4 row, 4 column matrices into which data is to be read.

The combination of ordinary BASIC statements and MAT statements makes BASIC very powerful. In addition to having both a DIM statement and a declaration of current dimension, the MAT statements should also be used with care. For example, a matrix product MAT C=A*B may be illegal for either of two reasons: A and B may have such dimensions that the product is not defined, or C may have the wrong dimensions for the answer. In either case the DIMENSION ERROR message is received.

SAMPLE MATRIX PROGRAMS

The following two programs illustrate some of the capabilities of the MAT statements. In the first program, the values for M and N are read. Using these two values as indices, statement 120 sets the dimensions for matrices A, B, D, and G. The values for the elements of these four matrices are read, then:

The dimensions of matrix C are specified and the elements set to zero (Line 130).

Matrix A is printed (Line 150).

Matrix B is printed (Line 170).

The sum of matrices A and B is calculated and stored in C (Line 180).

Matrix C is printed (Line 200).

The dimensions for matrix F, a vector, are set and the elements set to zero (Line 210).

The product of matrices C and D is computed and stored in F (Line 220).

The dimensions for matrix H (single value) are specified and the elements set to zero (Line 230).

Finally, the product of matrices G and F is found, stored in H and printed (Lines 240,260).

MATRIX PROGRAM EXAMPLE 1

```
100 DIM A(5,5),B(5,5),C(5,5),D(5,5)
105 DIM E(5,5),F(5,5),G(5,5),H(5,5)
110 READ M,N
120 MAT READ A(M,M),B(M,M),D(M,N),G(N,M)
130 MAT C=ZER(M,M)
140 PRINT "MATRIX A OF ORDER";M
150 MAT PRINT A;
160 PRINT "MATRIX B OR ORDER";M
170 MAT PRINT B;
180 MAT C=A+B
190 PRINT "      C=A+B"
200 MAT PRINT C;
210 MAT F=ZER(M,N)
220 MAT F=C*D
230 MAT H=ZER(N,N)
240 MAT H=G*F
250 PRINT "      H"
260 MAT PRINT H;
270 DATA 3,1
280 DATA 1,2,3,4,5,6,7,8,9,9,8,7,6,5,4,3,2,1,1,2,3,3,2,1
999 END
?RUN
```


MATRIX A OF ORDER 3

1 2 3

4 5 6

7 8 9

MATRIX B OF ORDER 3

9 8 7

6 5 4

3 2 1

C=A+B

10 10 10

10 10 10

10 10 10

H

360

MATRIX PROGRAM EXAMPLE 2

In this second program, a value N is read that determines the order of the Hilbert matrix segment to be computed, stored, and printed. Next the matrix is inverted and printed. Finally the Hilbert matrix is multiplied by its own inverse, and the resulting product matrix is printed. Notice that line 290 specified N as equal to 2 to produce the first three matrices of order 2, and later returns to read in the data '3', which redimensions to a larger array--larger than 2, but smaller than the original 20--and produces more output. The final case uses a value of 4 for the order.

```
100 DIM A(20,20),B(20,20),C(20,20)
110 READ N
120 MAT A=CON(N,N)
130 MAT B=CON(N,N)
140 MAT C=ZER(N,N)
150 FOR I=1 TO N
160 FOR J=1 TO N
170 LET A(I,J)=1/(I+J-1)
```

```

180 NEXT J
190 NEXT I
200 PRINT "HILBERT MATRIX OF ORDER";N
210 MAT PRINT A;
220 MAT B=INV(A)
230 PRINT "INVERSE OF HILBERT MATRIX OF ORDER";N
240 MAT PRINT B;
250 MAT C=A*B
260 PRINT "HILBERT MATRIX TIMES ITS OWN INVERSE ORDER";N
270 MAT PRINT C;
280 GO TO 110
290 DATA 2,3,4
999 END
?RUN

```

HILBERT MATRIX OF ORDER 2

1	0.50000
.50000	0.33333

INVERSE OF HILBERT MATRIX OF ORDER 2

4.00000	-6.00000
-6.00000	12.0000

HILBERT MATRIX TIMES ITS OWN INVERSE ORDER 2

1	0
5.82077E-11	1.00000

HILBERT MATRIX OF ORDER 3

1	0.50000	0.33333
0.50000	0.33333	0.25000
0.33333	0.25000	0.20000

INVERSE OF HILBERT MATRIX OF ORDER 3

9.00000	-36.0000	30.0000
-36.0000	192.000	-180.000
30.0000	-180.000	180.000

HILBERT MATRIX TIMES ITS OWN INVERSE ORDER 3

1.00000	1.86265E-09	-3.72529E-09
2.32831E-10	1.00000	-9.31323E-10
-1.16415E-10	-9.31323E-10	1.00000

HILBERT MATRIX OF ORDER 4

1	0.50000	0.33333	0.25000
0.50000	0.33333	0.25000	0.20000
0.33333	0.25000	0.20000	0.16667
0.25000	0.20000	0.16667	0.14286

INVERSE OF HILBERT MATRIX OF ORDER 4

16.0000	-120.000	240.000	-140.000
-120.000	1200.00	-2700.00	1680.00
240.000	-2700.00	6480.00	-4200.00
-140.000	1680.00	-4200.00	2800.00

HILEERT MATRIX TIMES ITS OWN INVERSE ORDER 4			
1.00000	4.47035E-08	-1.49012E-08	6.70552E-08
2.09548E-09	1.00000	8.19564E-08	-1.11759E-08
1.16415E-10	9.31323E-09	1.00000	-3.72529E-09
5.82077E-10	-3.72529E-09	1.11759E-08	1.00000

OUT OF DATA 110

CHAPTER 7 - EXAMPLES OF ADVANCED BASIC PROGRAMS

Following are two sample programs illustrating the use of many of the advanced capabilities of BASIC. The first program is developed as an inventory case problem, and utilizes a BCD file.

INVENTORY PROBLEM

Mr. Swift, a storekeeper, would like to know how any five items in his store are selling in any given month. He would like a permanent file of the items that were sold each week over a four week period. He also wishes to update his file at the end of each week, and he may or may not want a complete written record of his sales. However, he may want a written report at any time during the month.

The record should consist of an easy to read table listing the items and the number sold in each week. The table should also show the total number of items for each week, the total number of each sold to date, and the total number of all items sold to date.

The five items that Mr. Swift would like to check are salt, pepper, sugar, nutmeg, and coffee. The month is March.

The following program results from Mr. Swift's requirements. The program is explained by remarks included in it.

```
100 OPEN %1,"STOCK"
110 DIM A(5,4)
120 FOR I=1 TO 6
130 READ A$(I)
140 NEXT I
160 REM W=INITIAL PASS FLAG,P=PRINTOUT FLAG
170 REM W=0 INITIAL PASS, P=0 PRINTOUT DESIRED
180 READ W,P
190 IF W<0 THEN 300
210 REM FOR THE INITIAL PASS,WRITE 3 ZEROES. THERE IS NO
220 REM DATA INITIALLY, AND SOMETHING MUST BE WRITTEN
230 REM BEFORE IT CAN BE READ.
240 SCRATCH %1
250 WRITE %1,0;0;0
```

```

260 RESTORE %1
280 REM READ IN DATA WRITTEN INTO THE FILE FROM
285 REM PREVIOUS WEEKS
290 REM X..ITEM,Y..WEEK,A(X,Y)..NUMBERS OF ITEMS
295 REM SOLD TO DATE
300 READ %1,X,Y,Z
301 IF END %1 THEN 350
302 IF X GEQ 6 OR X <0 THEN 2000
304 IF Y GEQ 5 OR Y <0 THEN 2000
306 IF X EQ 0 OR Y EQ 0 GOTO 300
310 A(X,Y)=Z
320 GOTO 300
340 REM READ DATA FOR THIS WEEK OR DATA MISSED PREVIOUSLY
350 READ X,Y,Z
360 IF X<0 GOTO 430
380 REM WHEN X IS NEGATIVE THE DATA READ IS FINISHED.
390 LET Z(X,Y)=A(X,Y)+Z
400 GOTO 350
420 REM WRITE THE UPDATED INFO BACK TO THE PERMANENT FILE
430 SCRATCH %1
440 FOR X=1 TO 5
450 FOR Y=1 TO 4
460 WRITE %1,X;Y;A(X,Y);
470 NEXT Y
480 NEXT X
500 REM IS A PRINTOUT WANTED
510 IF P<0 THEN 870
530 REM PRINT THE MONTH
540 PRINT A$(6)
550 PRINT " "
560 PRINT TAB(10);
570 REM PRINT THE COLUMN HEADER FOR EACH WEEK
580 FOR I=1 TO 4
590 PRINT USING 610,I,
600 NEXT I
610: #####
620 PRINT USING 630
630: TOTALS
640 PRINT " "
660 REM BEGIN TO GENERATE THE TABLE OF VALUES.
670 FOR I=1 TO 5
680 PRINT A$(I);TAB(10);
690 FOR J=1 TO 4
700 REM SUM OF EACH ITEM FOR THE ELAPSED WEEKS.
710 T(I)=T(I)+A(I,J)
720 REM SUMS FOR EACH WEEK
730 LET S(J)=S(J)+A(I,J)
740 PRINT USING 610,A(I,J),
750 NEXT J
760 PRINT USING 610,T(I)
770 NEXT I
780 PRINT " "
790 PRINT "TOTAL";TAB(10);

```

```

800 REM PRINT SUBTOTALS FOR EACH WEEK AND
810 REM THE TOTAL FOR THE PERIOD.
820 FOR K=1 TO 4
830 PRINT USING 610,S(K),
840 S=S+S(K)
850 NEXT K
860 PRINT USING 610,S
870 STOP
880 REM 1..SALT,2..PEPPER,3..SUGAR,4..NUTMEG,5..COFFEE
890 DATA SALT,PEPPER,SUGAR,NUTMEG,COFFEE,MARCH
1000 DATA -1,0
1010 DATA -1 ,0,0
2000 PRINT "INVALID ITEM",X,Y,Z
2005 GOTO 300
8000 END
?RUN

```

Suppose that two weeks have passed and Mr. Swift wants a record of his sales to date. He runs the program with data in line 1000 as shown above. The -1 specifies not the first week, and the 0 specifies a printout of the file data. Also he enters data in line 1010 as shown above. The -1 indicates the termination of data, and the two final zeroes are dummy data put in to satisfy the read statements. See the following results.

MARCH					
	1	2	3	4	TOTALS
SALT	3	4	0	0	7
PEPPER	7	5	0	0	12
SUGAR	4	3	0	0	7
NUTMEG	8	7	0	0	15
COFFEE	2	9	0	0	11
TOTAL	24	28	0	0	52

At the end of the third week, an update and a printout are required. All Mr. Swift must do is replace line number 1010, as shown on the next page. The BCD data file is updated, and the printout shows the entries for the third week and the resulting changes in the totals.

```
1010 DATA 1,2,7,2,3,5,3,3,1,4,3,5,5,3,12,-1,0,0
?RUN
```

MARCH

	1	2	3	4	TOTALS
SALT	3	4	7	0	14
PEPPER	7	5	5	0	17
SUGAR	4	3	1	0	8
NUTMEG	8	7	5	0	20
COFFEE	2	9	12	0	23
TOTAL	24	28	30	0	82

VALUE OF E TO 1000 PLACES

The following program asks for a number less than 1000 and then calculates the value of E (base of natural logarithms) to that many decimal places. The string variable C\$ is used to generate the individual digits of the result. The inner loop between lines 190 and 240 gives one more digit of the result each time the loop is entered. Line 250 converts an integer digit M to a character in the output line. Line 275 causes the line to be printed after the FOR loop starting at 170 has been exhausted. Each pass through the loop starting at 170 produces one line of output with up to 25 digits of the result.

```
5 C$(1)="0"
10 C$(2)="1"
15 C$(3)="2"
20 C$(4)="3"
25 C$(5)="4"
30 C$(6)="5"
35 C$(7)="6"
40 C$(8)="7"
45 C$(9)="8"
50 C$(10)="9"
60 PRINT "HOW MANY DECIMAL PLACES";
70 INPUT P
80 IF P GTR 1000 GOTO 60
90 X=0
100 DIM A(1000)
110 PRINT " THE VALUE OF E IS APPROXIMATELY 2."
120 FOR I=2 TO 1000
130 A(I)=1
140 NEXT I
150 A(1)=0
```

```

160 FOR I= 1 TO INT (P/25+1)
170 FOR N=1 TO 25
180 M=0
190 FOR J=2 TO 1000
200 K=1002-J
210 L=A(K)*10+M
220 M = INT(L/K)
230 A(K)=L-M*K
240 NEXT J
250 PRINT C$(M+1);
254 X=X+1
256 IF X GEQ P THEN 300
260 A(1)=0
270 NEXT N
275 PRINT
280 NEXT I
300 PRINT
1000 END
?RUN

```

HOW MANY DECIMAL PLACES?500

THE VALUE OF E IS APPROXIMATELY 2.

```

7182818284590452353602874
7135266249775724709369995
9574966967627724076630353
5475945713821785251664274
2746639193200305992181741
3596629043572900334295260
5956307381323286279434907
6323382988075319525101901
1573834187930702154089149
9348841675092447614606680
8226480016847741185374234
5442437107539077744992069
5517027618386062613313845
8300075204493382656029760
6737113200709328709127443
7470472306969772093101416
9283681902551510865746377
2111252389784425056953696
7707854499699679468644549
0598793163688923009879312

```


APPENDICES

APPENDIX A - TELETYPE OPERATION

The Control Data 3300 computer at Oregon State operates under a time-sharing operating system called OS-3 ("Oregon State Open Shop Operating System").

Each user must first establish contact with the computer through the facilities of the operating system. In local jargon, this is called "Logging On." A brief, but hopefully adequate description of how to log on and off will be given here. For a more complete treatment of OS-3, the reader may consult ccm-71-07, "Primer for OS-3 Users".

Before you can log on to the system, you must have a currently valid six-digit job number/four-character user code combination. To obtain information regarding these access codes, visit or call Room 126 in the basement of the Computer Center (phone 754-2494).

When you have obtained a job number/user code and have located a teletype which is (or can be) connected to the system at OSU, follow these steps to log on.

- 1) Turn the MODE SELECT knob on the front right-hand corner of the machine to LINE position.
- 2) Depress the "CNTL" key (leftmost key, third row from top) and the A key simultaneously.
- 3) The computer (OS-3 actually) will type "#" and wait about 20 seconds for you to enter your job number, a comma, your user code, and then terminate the line with a carriage return (CR key, second key from right end, second row down).
- 4) If you do all this within 20 seconds, OS-3 will type masking characters over what you typed, then print a message giving the data, time, and number of your terminal (or perhaps a message "ILLEGAL JOB/USER NUMBER" if you mistyped a character).

- 5) If you get the ILLEGAL NUMBER message, or no message at all, repeat steps 2 and 3. Otherwise, OS-3 will have typed another "#" - its trademark - and you may now enter BASIC by typing BASIC followed by a carriage return.

When you wish to conclude your run, you must once again establish communication with the OS-3 control mode routine (trademark "#"). This may be accomplished by again depressing the CNTL and A keys simultaneously. The control mode routine will respond with "#", and you may type LOGOFF and a carriage return to end your session with the computer. Note that unless you have taken steps to save your program, it will be lost when you logoff.

APPENDIX B

FORMS OF BASIC STATEMENTS

<u>STATEMENT</u>	<u>SYNTAX</u>
LET	LET<variable>=<expression> (LET is optional)
READ	READ<variable>,<variable>,...,<variable> READ % LUN<variable>,<variable>,...,<variable>
MAT READ	MAT READ<matrix variable>,<matrix variable>,..., <matrix variable>
DATA	DATA<number>,<number>,...,<number>
PRINT	PRINT<list> <list> may include: a variable, an expression, a message in quotes, or any combination of these separated by commas or semi-colons. Example: 10 PRINT A, B, C, "THE VALUE OF X IS", A+B*COS(C) PRINT % LUN,<list>
MAT PRINT	MAT PRINT<matrix variable>,<matrix variable>,..., <matrix variable> MAT PRINT:<filename>:<matrix variable>, <matrix variable>,...,<matrix variable>
INPUT	INPUT<variable>,<variable>,...,<variable>
WRITE	WRITE:<filename>:<variable>,<variable>,..., <variable>
MAT WRITE	MAT WRITE:<filename>:<matrix variable>,<matrix variable>,...,<matrix variable>
GO TO	GO TO<line number>
ON	ON<expression>GO TO<line number>
IF - THEN	IF<expression><relational symbol><expression> THEN<line number> IF<expression><relational symbol><expression> GO TO<line number>

FOR	FOR<unsubscripted variable>=<expression>TO	[NOTE: FOR and NEXT are used <u>in pairs</u> <u>only</u> to delimit a loop to be repeatedly executed. FOR-NEXT pairs may occur within other such pairs.
	<expression>STEP<expression>	
NEXT	NEXT<unsubscripted variable>	
END	END	
STOP	STOP	
DEF	DEF FN<letter>(<unsubscripted variable>)=<expression>	
GOSUB	GOSUB<line number>	
RETURN	RETURN	
DIM	DIM<letter>(<integer>),<letter>(<integer>,<integer>)...	
OPEN	OPEN % <lun>,"name"	
IFEND	IFEND % LUN	
CLOSE	CLOSE % LUN	
REM	REM <any string of characters>	
RESTORE	RESTORE	
	RESTORE % LUN	
SCRATCH	SCRATCH % LUN	
MAT	MAT<matrix variable>=<matrix variable>+<matrix variable>	
	MAT<matrix variable>=<matrix variable>-<matrix variable>	
	MAT<matrix variable>=<matrix variable>*<matrix variable>	
	MAT<matrix variable>= INV<matrix variable>	
	(Invert matrix)	
	MAT<matrix variable>= TRN<matrix variable>	
	(Transpose matrix)	
	MAT<matrix variable>= (<number>)*<matrix variable>	
	MAT<matrix variable>=ZER	
	(Fill matrix with zeros)	
	MAT<matrix variable>=CON	
	(Fill matrix with ones)	
	MAT<matrix variable>=IDN	
	(Set up identity matrix)	
	MAT<matrix variable>=<matrix variable>	

BASIC FUNCTIONS

Available functions are:

<u>Function</u>	<u>Purpose</u>	
SIN(X)	Sine of X	} X must be in radians
COS(X)	Cosine of X	
TAN(X)	Tangent of X	
ATN(X)	Arctangent of X	
EXP(X)	Natural exponential of X	
ABS(X)	Absolute value of X	
LOG(X)	Natural logarithm of X	
SQR(X)	Square root of X	
SGN(X)	Sign of X (-1 if X<0, 0 if X=0, +1 if X>0)	
RND(X)	Generate random number	
INT(X)	Integer part of X	
TAB(X)	Used in a PRINT statement. Tabs over to the Xth print position. If the current print position is greater than X it is ignored.	

APPENDIX C

ERROR MESSAGES

Because most programs under development contain errors, a series of error messages is included in BASIC. Some of the messages are received during compilation and others during execution of a program. Many of the messages not only identify the type of error, but indicate the line number where the error occurred. In the following table, XXX denotes a line number.

During execution, some messages occur that do not stop execution, but inform the user of irregular conditions existing in identified lines of the program. Other messages, however, point out more serious errors that cause execution to stop.

COMPILATION ERRORS

<u>MESSAGE</u>	<u>MEANING</u>
DIMENSION ERROR XXX	An array has previously been dimensioned.
END IS NOT LAST	END statement encountered before the last of the program.
FOR WITHOUT NEXT XXX	There is no NEXT statement that matches FOR statement.
ILLEGAL FORMULA XXX	Perhaps the most common error message. May indicate missing parentheses, illegal variable names, missing multiplication signs, illegal numbers, or almost any syntax error. Check the statement thoroughly.
INCORRECT FORMAT XXX	Error in data for a DATA statement, improper subscript format, or incorrect statement syntax.
LABEL IS UNDEFINED XXX	Line XXX contains a reference to a label that does not exist.
NO END STATEMENT	The END statement is missing from the program.

NOT ENOUGH NEXTS	A NEXT statement is missing. This message can occur in conjunction with NOT MATCH WITH FOR XXX.
NOT MATCH WITH FOR XX	There is an incorrect NEXT statement, perhaps with a wrong variable given. Check also for incorrectly nested FOR statements.
PROGRAM TOO LONG	Either the program is too long, or the amount of space reserved by the DIM statements is too much, or both. Cut the length of the program, reduce the length of printed labels, or reduce the size of the lists and tables, or reduce the number of simple variables.
SYSTEM OR BASIC ERROR	Probably due to an error in the BASIC compiler. Please notify Computer Center. If possible provide documentation so that the problem can be fixed more readily.
UNDEFINED FUNCTION XXX	The function referenced at line XXX has not been defined with a DEF statement.
<u>COMMAND MODE ERRORS</u>	
FILE DOES NOT EXIST	The file name specified in a FIN command does not exist.
ILLEGAL COMMAND	The command just typed in is not one of the legal commands specified in Appendix B.
LINE NOT FOUND	The line specified in a list command does not exist.
PARAMETER ERROR	A parameter for the BASIC command just given is in error.
<u>EXECUTION ERRORS</u>	
ABSOLUTE VALUE RAISED TO POWER XXX	A computation of the form $(-3)^{2.7}$ has been attempted. The computer supplies $(\text{ABS}(-3)^{2.7})$ and continues. Note: $(-3)^3$ is correctly computed to give -27.
ARRAY BOUNDS ERROR XXX	A subscript has been called for that lies outside the range specified in the DIM statement, or, if statement applies outside the range 1 through 10 or is 0. The program halts.

BAD IMAGE XXX	There are syntax errors in the image statement referenced by line number XXX, or an attempt has been made to put numeric data in an alphanumeric field, or alphanumeric data in a numeric field.
DIMENSION ERROR XXX	A dimension inconsistency has occurred. The program stops.
DIVIDE FAULT XXX	A division by zero has been attempted. The computer supplies 0 and continues running the program.
EXP TOO LARGE XXX	The argument for the EXP function is greater than 709.089.
EXPONENT FAULT XXX	A number larger than about 8.98847E307 has been generated. The computer supplies 0 and continues running the program. A number smaller in absolute size than about 10E-307 has been generated. The computer supplies zero and continues. In many circumstances, underflow is permissible and may be ignored.
FILE ALREADY OPEN XXX	The logical unit specified in an OPEN statement is already open.
FILE IS BUSY	The file specified in a FIN, FILE, or OPEN is currently open at another terminal or in a currently running batch job. Also, if you open a file within the IOS subsystem and then transfer to BASIC with the file still open, BASIC will give the FILE IS BUSY message. This is not the case if going from the EDIT subsystem to BASIC since EDIT will automatically close any open files when transferring to any other subsystem.
FILE NOT OPEN XXX	The file OPEN statement is missing for this logical unit.
ILLEGAL LUN XXX	The logical unit referenced at line XXX is not in the range of 1 to 20.
INPUT DATA NOT IN CORRECT FORMAT XXX	Data for an INPUT or a READ statement is not in the correct format.
INSUFFICIENT SAVE FILE SPACE	Not enough saved file space available for a file.

LOG OF NEG NUMBER XXX	The program has attempted to calculate the logarithm of a negative number. The computer supplies the logarithm of the absolute value and continues.
LOG OF ZERO XXX	The program has attempted to calculate the logarithm of zero. The computer supplies -8.98847E307 and continues.
MATRIX MUST BE SQUARE	Program tried to invert a non-square matrix at line XXX.
NEARLY SINGULAR MATRIX XXX	The INV operator in MAT has encountered a matrix with zero or nearly zero pivotal elements. The matrix being inverted is singular or nearly so. Note, however, that this check is not completely reliable. For instance, this message need not occur even if the inverse is meaningless, as with high order Hilbert matrices. If this error occurs, the program stops.
NUMBER TOO LARGE XXX	The result of an exponentiation is greater than 8.988465E307. The value 8.988465E307 used as a result.
OUT OF DATA XXX	A READ statement has been encountered for which there is no DATA. If this means a normal end of your program, ignore the message. Otherwise, it means that insufficient data has been supplied. In either case, the program halts.
ON ERROR XXX	The range of an ON--GOTO statement is incorrect. Example: ON X GOTO 10,20,30. When the integer value of X is either minus, zero, or greater than 3, the expression is out of range.
READING FROM OUTPUT UNIT XXX	An attempt has been made to read from a write mode file. Indicates a logic error or no RESTORE statement encountered before read mode activity.
RETURN BEFORE GOSUB XXX	A RETURN has been encountered before the first GOSUB in the program. Note: BASIC does not require the GOSUB to have an earlier statement number--only to execute a GOSUB before executing a RETURN. The program stops.

SQUARE ROOT OF A NEGATIVE
NUMBER XXX

The program has attempted to extract the square root of a negative number. The computer supplies the square root of the absolute value and continues.

TOO MANY GOSUBS XXX

Too many GOSUB calls have been made. This refers to the number of nested GOSUB calls. The level of nesting is counted up by one for each GOSUB that is executed and is counted down by one for each RETURN executed.

UNCHECKED END OF FILE XXX

An attempt has been made to write into a read mode file. Indicates a logic error or no SCRATCH statement encountered before write mode activity.

/X/ GT 2↑36-1 XXX

The argument for a SIN, COS, or TAN function is too big. A value of zero is supplied and execution continues.

ZERO TO A NEGATIVE POWER XXX

A computation of the form $0 \uparrow (-1)$ has been attempted. The computer supplies + (about 8.98847E307) and continues.

APPENDIX D

COMPARISON ORDER FOR BASIC CHARACTERS

Collating Sequence	Internal Octal Code	Terminal	Line Printer
00	60	Space	Blank
01	12	:	:
02	13	=	=
03	14	/	≠
04	15	&	≤
05	16	%	%
06	17	[[
07	20	+	+
08	32	<	<
09	33	•	•
10	34))
11	35	"	≥
12	36	#	¬(not)
13	37	;	;
14	40	-	-
15	52		v(or)
16	53	\$	\$
17	54	*	*
18	55	↑	↑

Collating Sequence	Internal Octal Code	Terminal	Line Printer
19	56	@	↓
20	57	>	>
21	61	/	/
22	72]]
23	73	,	,
24	74	((
25	75	←	→
26	76	\	=
27	77	?	Δ (and)
28	00	0	0
29	01	1	1
30	02	2	2
31	03	3	3
32	04	4	4
33	05	5	5
34	06	6	6
35	07	7	7
36	10	8	8
37	11	9	9

Collating Sequence	Internal Octal Code	Terminal	Line Printer
38	21	A	A
39	22	B	B
40	23	C	C
41	24	D	D
42	25	E	E
43	26	F	F
44	27	G	G
45	30	H	H
46	31	I	I
47	41	J	J
48	42	K	K
49	43	L	L
50	44	M	M
51	45	N	N
52	46	O	O
53	47	P	P
54	50	Q	Q
55	51	R	R
56	62	S	S

Collating Sequence	Internal Octal Code	Terminal	Line Printer
57	63	T	T
58	64	U	U
59	65	V	V
60	66	W	W
61	67	X	X
62	70	Y	Y
63	71	Z	Z